9

Requirements Engineering

LEARNING OBJECTIVES

- To understand that requirements engineering is a cyclical process involving four types of activity: elicitation, specification, validation, and negotiation
- To appreciate the role of social and cognitive issues in requirements engineering
- To be able to distinguish a number of requirements elicitation techniques
- To be aware of the contents of a requirements specification document
- To know various techniques and notations for specifying requirements
- To know different ways to structure a set of requirements

This chapter covers requirements engineering, the first major phase in a software development project. The most challenging and difficult aspect of requirements engineering is to get a complete description of the problem to be solved. We discuss a number of techniques for eliciting requirements from the user. Following elicitation, these requirements must be negotiated, validated, and documented.

> *The hardest single part of building a system is deciding what to build*
> (Brooks, 1987)

The requirements engineering phase is the first major step towards the solution of a data processing problem. During this phase, the user's requirements with respect to the future system are carefully identified and documented. These requirements concern both the functions to be provided and a number of additional requirements, such as those regarding performance, reliability, user documentation, user training, cost, and so on. During the requirements engineering phase we do not yet address the question of *how* to achieve these user requirements in terms of system components and their interaction. This is postponed until the design phase.

A requirement is 'a condition or capability needed by a user to solve a problem or achieve an objective' (IEEE610, 1990). The 'user' alluded to in this definition may be an end user of the system, a person behind the screen. However, it may also denote several classes of indirect users, such as people who do not themselves turn the knobs but rather use the information that the system delivers. It may also denote the client (customer) who pays the bill. During requirements engineering, different types of user may be the source of different types of requirements. Hopefully, the end users will be the main source of information regarding the functional, task-related requirements. Other requirements, e.g. those that relate to security issues, may well be phrased by other stakeholders.

The word 'requirement' suggests that, once stated, it **has** to be met. In reality, this hardly ever is the case. Most requirements are negotiable. Time to market, cost, conflicting quality requirements, and conflicting needs of stakeholders all lead to a situation where tradeoffs may have to be made.

The result of the requirements engineering phase is documented in the **requirements specification**. The requirements specification reflects the mutual understanding of the problem to be solved between the analyst and the client. It is the basis for a contract, be it formal or informal, between the client of the system and the development organization. Eventually, the system delivered will be assessed by testing its compliance with the requirements specification.

The requirements specification serves as a starting point for the next phase, the design phase. In the design phase the architecture of the system is devised in terms of system components and interfaces between those components. The design phase results in a specification as well: a precise description -- preferably in some formal language -- of the design architecture, its components, and its interfaces.

The notion 'specification' thus has several meanings. To prevent confusion, we will always use the prefix 'requirements' if it denotes the result of the requirements engineering phase.

To make matters worse, the phase in which the user's requirements are analyzed and documented is also sometimes called specification. We feel this to be somewhat of a misnomer and will not use the term as such.

We use the term **requirements engineering** rather than the narrower notion of **requirements analysis** to emphasize that it is an iterative and co-operative process of analyzing a problem, documenting the resulting observations, and checking the accuracy of the understanding gained. Requirements engineering not only involves technical concerns of how to represent the requirements. Social and cognitive aspects play a dominant role as well.

Requirements engineering and design generally cannot be strictly separated in time. In some cases, the requirements specification is very formal and can be viewed as a high-level design specification of the system to be built. Often, a preliminary design is done after an initial set of requirements has been determined. Based on the result of this design effort, the requirements specification may be changed and refined. This type of iteration also occurs when prototyping techniques are being used. In pure agile development projects, requirements emerge concurrently with an up-and-running system. Well-known techniques such as data flow diagrams and UML class diagrams are used to structure and document both requirements specifications and designs.

It is only for ease of presentation that the requirements engineering and design phases are strictly separated and treated consecutively in this book.

During requirements engineering, a number of quite different matters are being addressed. Let us look at an example and consider the (hypothetical) case of a university's library automating its operation. We will start with the library containing a number of cabinets. These cabinets hold a huge number of cards, one per book. Each card contains the names of the authors, the book title, ISBN, publication year, and other useful data. The cards are ordered alphabetically by the name of the first author of each book.

This ordering system in fact presents major problems as it only works well if we know the first author's name. If we only know the title, or if we are interested in books on a certain topic, the author catalog is of little or no help.

A software solution seems obvious. If we store the data for each book once in a database, we may subsequently sort the entries in many different ways. Appropriate tools can enable the user to search the database interactively. By providing internet access to the database, service can be greatly enhanced.

During the requirements engineering phase, a number of user requirements will be raised. Some of those requirements will concern updating the database, that is adding, deleting and changing records. Others will concern functions to be provided to ordinary members of the library, such as:

– Give a list of all books written by X;

- Give a list of all books whose title contains Y;
- Give a list of all books on topic Z;
- Give a list of all books that arrived after date D.

It is expedient to try somehow to group user requirements into a few categories, ranging from 'essential requirements' to 'nice features'. As noted in chapter 3, users tend to have difficulties in articulating their real needs. Chances are, then, that much effort is spent on realizing features which later turn out to be mere bells and whistles. By using a layered scheme in both the formulation of user requirements and their subsequent realization, some of the problems that beset software development projects can be circumvented. In our library system example, for instance, the requirement 'Give a list of all books that arrived after date D' could be classified as a nice feature. Service is not seriously degraded if this function is not provided, since we may temporarily place the acquisitions on a dedicated shelf.

It is also possible to try to predict a number of future requirements, which will not be implemented in the present project. It is, however, sensible to pay attention to these matters at an early stage, so that they can be accommodated during the design of the system. Possible future requirements of our library system could include such things as:

- Storing information about books that have been ordered but have not been received;
- Storing information about library members, such as their name and address, and the dates on which books are lent to them, which can then be used to generate a reminder notice for books not returned on time.

The above functions concern the use of the software by library members and library personnel. There are other stakeholders as well, though. For example, library management may wish to use the system to get information on member profiles in order to improve the title acquisition process.

Besides these requirements, which directly relate to the functions of the software to be delivered, a number of other matters should be addressed during the requirements engineering phase. For our library example, as a minimum, the following points have to be addressed:

- On which machine will the system be implemented, and which operating system will be used? If the data is to be stored in some DBMS, which (type of) DBMS is to be used? What type of access is to be used and how many access points will be supported?
- Which classes of users can be distinguished? In our example, both library personnel and library members will have to be served. What kind of knowledge do these users have? Will certain functions of the system be restricted to certain

classes of user? Normal library members will probably not be allowed to update the database or print the contents of the database.

- What is the size of the database and how is it expected to grow in the course of time? These factors influence both storage capacity needed and algorithms to be used. For a database containing several thousands of books, some not very efficient searching algorithm might suffice. For the Library of Congress, the situation is quite different, though.

- What response time should the system offer? A search request for a certain book will have to be answered fairly quickly. If the user has to wait too long for an answer, he will become dissatisfied and search the shelves directly. Related questions concern the interaction between response time and the expected number of question sessions per unit of time.

- How much will a system of this kind cost? In our library example, we should not only pay attention to the direct costs incurred by the software development effort. The cost of converting the information contained in the present file cabinets to a suitable database format should not be neglected. These, less visible, indirect costs may well outweigh the direct cost of designing and implementing the new system.

This relatively simple example already shows that it is not sufficient to merely list the functional requirements of the new system. The system's envisioned environment and its interaction with that environment should be analyzed as well.

In our example, this concerns the library itself, to start with. The consequences of introducing a system like this one can be much greater than it seems at first sight. Working procedures may change, necessitating retraining of personnel, changes in personnel functions and the overall organizational structure. Some members of staff may even become redundant. Checking whether membership fees have been paid might involve interfacing with the financial system, owned by another department.

In general, the setting up of an automated system may have more than just technical repercussions. Often, not enough attention is paid to these other repercussions. The lack of success of many software development projects can be traced back to a neglect of non-technical aspects.

In practice, the requirements engineering process often is more complex than sketched above:

- Ordinary library automation is a relatively well-known domain, where we may expect users to be able to articulate their requirements. But suppose our library system also has to support elderly people in their dealings with the world around them, including daily news, relevant government regulations, information about healthcare, and the like. For the latter type of support, a more agile approach seems more appropriate, where the requirements emerge as we go along, rather than being elicited up front.

- Much software developed today however is *market-driven* rather than *customer-driven*. For example, rather than developing a system for one specific library, we could develop a 'generic' library application. Requirements for this generic library application are created by exploring the library domain, while trade-offs between requirements are based on market considerations, product fit, and the like. We may decide that our system need not address the concerns of the Library of Congress (too small a market), while it should definitely interface with accounting system Y, since that system is widely used in university departments, and this is perceived to be an important market for our library application.

- For different reasons (cost, time to market, quality) we may want to employ commercial off the shelf (COTS) components in our library system. We then have to tradeoff our requirements against the possibilities offered by those COTS components.

Following Sommerville (2005), we distinguish four processes in requirements engineering:

- **Requirements elicitation** In general, the requirements analyst is not an expert in the domain being modeled. Through interaction with domain specialists, such as professional librarians, he has to build himself a sufficiently rich model of that domain. Thus, requirements elicitation is about *understanding* the problem. The fact that different disciplines are involved in this process complicates matters. In many cases, the analyst is not a mere outside observer of the domain to be modeled, simply eliciting facts from domain specialists. He may have to take a stand in a power struggle or decide between conflicting requirements, thereby actively participating in the construction of the domain of interest. Section 9.1 discusses various issues related to, and a number of techniques used in, requirements elicitation.

- **Requirements specification** Once the problem is understood, it has to be *described*. In section 9.2, we give guidelines for the contents of a requirements specification document. This document describes the product to be delivered, not the process of how it is developed. Project requirements are described in the project plan, discussed in chapter 2. The collection of requirements not only has to be documented, it also has to be managed during the course of a project. Quite a number of techniques exist for specifying requirements, ranging from very informal (natural language) to very formal (mathematical). Throughout this book, a number of such modeling techniques are discussed, such as object-oriented techniques in chapter 10, and techniques for specifying quality requirements in chapter 6. The design techniques discussed in chapter 12 are often also used for specifying requirements. Section 9.3 is confined to a discussion of some of the basic techniques for modeling requirements.

- **Requirements validation and verification** Once the problem is described, the different parties involved have to *agree upon* its nature. We have to ascertain that the correct requirements are stated (validation) and that these requirements are stated correctly (verification). Some verification and validation techniques that can be applied at this early stage are sketched in section 9.4.

- **Requirements negotiation** Usually, requirements have to be negotiated. Because of time constraints or other factors, a selection may have to be made from the list of requirements put forth. Or stakeholders may have conflicts that need to be resolved. Often, stakeholders have conflicting quality requirements whose impact can only be determined by looking at the software architecture. This is further dealt with in chapter 11.

Obviously, these processes involve iteration and feedback. In document-driven approaches, these iterations preceed design and implementation. In agile processes, design and implementation is part of the iteration and feedback loop. In either case, there is a central repository in which the requirements are documented. The major interactions are shown in figure 9.1.

The emphasis in our discussion of requirements engineering will be on modeling the external behavior of the system, i.e. all those parts and aspects of the system that *end users* consider important. Other views are relevant as well, for instance, a model which highlights the way the system supports the business, or a model which indicates how a system is deployed on a collection of hardware devices. Some of these other views are discussed in chapter 11.

9.1 Requirements Elicitation

In chapter 1, the first part of the software life cycle was depicted as shown in figure 9.2. The fact that the text 'requirements specification' is placed in a rectangle suggests, not unjustly, that it concerns something very concrete and explicit. The 'problem' is less well defined, less clear, even fuzzy in many cases. The primary goal of the requirements engineering phase is to elicit the contours and constituents of this fuzzy problem. This process is also known as **conceptual modeling**.

During requirements engineering we are modeling part of reality. The part of reality in which we are interested is referred to as the **universe of discourse** (UoD). Example UoDs are a library system, a factory automation system, an assembly line, an elevator system.

The model constructed during the requirements engineering phase is an **explicit conceptual model** of the UoD. The adjective 'explicit' indicates that the model must be able to be communicated to the relevant people (such as analysts and users). To this end it should contain all relevant information from the UoD. One of the persistent problems of requirements analysis and, for that matter, analysis in general, is to account for all of the relevant influences and leave out irrelevant details.

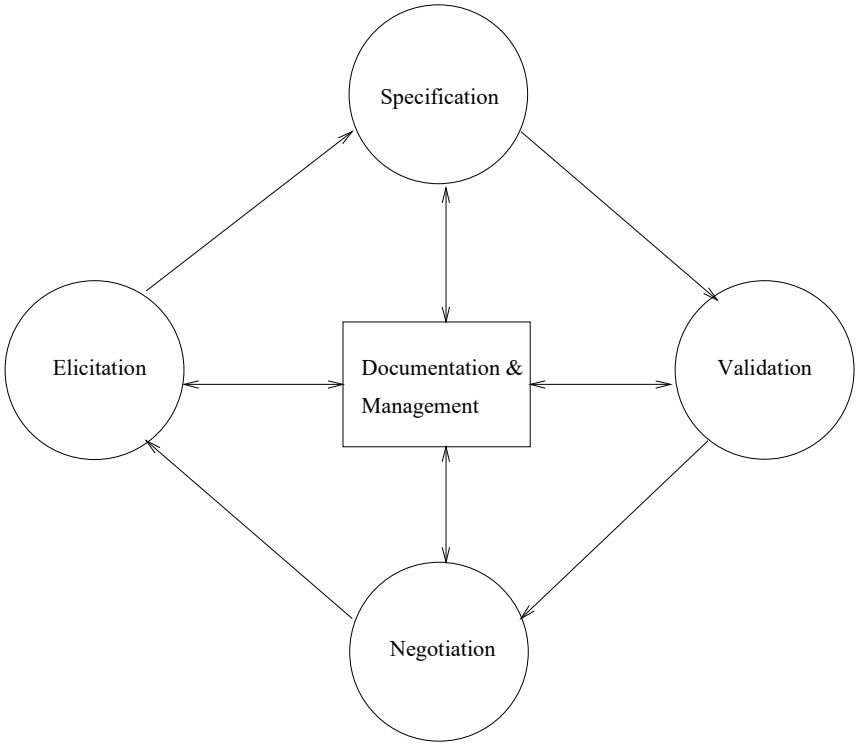

Figure 9.1 A framework for the requirements engineering process (adapted from (Sommerville, 2005))

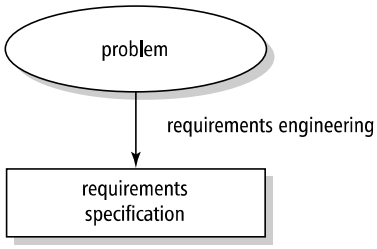

Figure 9.2 The first part of the software life cycle

In our library example we could easily have overlooked the fact that in a number of cases the author's name as it appears on the cover of a book is not the 'canonical' author's name. This phenomenon occurs in particular with authors from countries that use non-Latin scripts. The transcription of the Russian name ЧЕХОВ reads 'Chekhov' in English and 'Tsjechow' in Dutch. In such cases, librarians want to include the author's name twice: once the name is spelled as it appears on the book, and once the name is spelled as it is used in the various search processes. An answer to a question like 'which books by Chekhov does our library possess?' should also inform us about the non-English titles.

Subtle mismatches between the analyst's notion of terms and concepts and their proper meaning within the domain being modeled can have profound effects. Such mismatches can most easily occur in domains we already 'know', such as a library. An illuminating discussion of potential problems in (formally) specifying requirements of a library system can be found in (Wing, 1988). Problems noted include:

- A library employee may also be a member of the library, so the two sets of system users are not disjoint;
- There is a difference between a book (identified by its ISBN) and the (physical) copies of a book owned by the library;
- It is not sufficient to simply denote the status of a book by a boolean value present/not present (i.e. lent out). For instance, a book or, more properly, a copy of a book, may be lost, stolen, or in repair.

People involved in a UoD have an **implicit conceptual model** of that UoD. An implicit conceptual model consists of the background knowledge shared by people in the UoD. The fact that this knowledge is shared gives rise to 'of course' statements by people from within the UoD, because this knowledge is taken for granted. ('Of course, a copy of a book is not the same as a book.') Part of the implicit conceptual model is not verbalized. It contains tacit knowledge, knowledge that is skillfully applied and functions in the background. Finally, an implicit conceptual model contains habits, customs, prejudices and even inconsistencies.

During conceptual modeling, an implicit conceptual model is turned into an explicit one. In doing so, the analyst is confronted with two types of problem: analysis problems and negotiation problems. Analysis problems arise from the fact that part of the implicit conceptual model is not verbalized, that the implicit conceptual model evolves with time, that the user and analyst talk a different language, and that the implicit conceptual model cannot be completely codified. Negotiation problems arise because people in the UoD may counteract the analysis process, because the implicit conceptual models of people in the UoD may differ, or because of opposing interests of people involved (such as library personnel versus their managers). Both types of problems are discussed below.

The problem to be addressed by the automated system arises from the user, a human. This person must be able to describe this problem in both a correct and

complete way. It must be communicated to a person who in general has a rather different background. The analyst often lacks a sufficiently profound knowledge of the application domain in which the problem originated. He has to learn the language of the application domain and become acquainted with its terminology, concepts and procedures. Especially in large projects, the application knowledge tends to be thinly spread amongst the specialists involved, which easily leads to integration and coordination difficulties.

In our earlier example, it is the librarian who has to express his wishes. It is possible that the inclusion of two author names ('Tsjechow' and 'Chekhov') is seen as an obvious detail which need not be brought forward explicitly. The analyst at the other side of the table may still get the impression that he has a complete picture of the system. This type of omission may have severe consequences.

> A number of years back a large automated air defense system was being developed in the US. During one of the final tests of this system, an alarm signal was issued. One of the computers detected an unknown missile. It turned out to be the moon. This possibility had not been thought of.

Eliciting correct and complete information is an important prerequisite for success. This turns out to be rather problematic in practice. Asking the prospective user what is wanted does not generally work. More often than not we get a rather incomplete and inaccurate picture of the situation. Important reasons for this are the human limitations for processing information, selecting information, and solving problems. These limited human capabilities are yet aggravated by such factors as:

- the complexity and variation in requirements that can be imposed upon software;
- the differences in background between the client, or user, and the software specialist.

In research on human information processing one often uses a model in which human memory consists of two components: a short-term memory in which information is being processed, and a long-term memory in which the permanent knowledge is stored. Short-term memory has a limited capacity: one often says that it has about seven slots. Long-term memory on the other hand has a very large capacity.

So, information is processed in a relatively small part of human memory. Long-term memory is thus accessed in an indirect way. In addition, humans also employ external memories when information is being processed: a blackboard, a piece of paper, etc.

If a person being interviewed during requirements engineering only uses his short-term memory, the limitations thereof may have an impact on the results. This may easily occur if no use is being made of external memories. Things can be forgotten, simply because our short-term memory has limited capacity.

Humans are also inclined to be prejudiced about selecting and using information. We are, in particular, inclined to let recent events prevail. In making up a requirements

specification, this leads to requirements bearing on the present situation, presently available information, recent events, etc.

Humans are not very capable of rational thinking. They will simplify things and use a model which does not really fit reality. Other limitations that influence our model of reality are determined by such factors as education, prejudice, practice, etc. This same kind of simplification occurs when software requirements are drawn up. And the result will be limited by the same factors.

We cannot always expect the user to be able to precisely state his requirements at an early stage. One reason for investigating the opportunities of automation is often because of a certain dissatisfaction with the present situation. One is not satisfied with the present situation and has the impression that automation will help. Whether this is true or not -- many data processing problems are organizational problems -- simply automating the present situation is not always the solution. Something different is wanted, though it is not clear what. Only when insight into the possibilities of automation is gained, will real requirements show themselves. This is one of the reasons for the sheer size of the maintenance problem. About half of the maintenance effort concerns adapting software to (new) requirements of the user. To counteract this trend, software development process models that acknowledge this learning process, such as prototyping, incremental development, and agile methodss are to be preferred over those that don't, i.e. the waterfall model and its variants.

Through a careful analysis, we may hope to build a sound perspective of user requirements and anticipate future changes. However, no matter how much time is spent in a dialog with the prospective users, future changes remain hard to foresee. We may even go one step further and stipulate that requirements will *never* be complete. In this respect, specifying requirements has much in common with weather-forecasting: there is a limit to how far the future can be predicted.

In a situation where the goal of a software development project is to improve an existing 'system', be it a manual process or a (partly) automated one, it is generally helpful to explicitly distinguish two modeling steps. In the first step, the current situation is modeled. Based on an analysis of the strengths and weaknesses of the current situation, the situation-to-be is next modeled. Business Process Redesign, in particular, stresses the distinction between these two modeling steps.

For the requirements engineering phase to be successful we need methods and techniques that try to bypass the difficulties sketched above. The degree to which powerful techniques are required depends on the experience of the people involved in the requirements engineering phase (both users and analysts) and the expertise of the analyst with the application domain. Section 9.1.2 discusses a number of techniques for requirements elicitation.

But before we discuss these techniques, we first elaborate in section 9.1.1 how different world views result in different approaches to requirements engineering. In section 9.1.3 we discuss how requirements relate to higher level goals, and how different viewpoints may result in different, and sometimes conflicting, sets of requirements. Following section 9.1.3 we discuss how to prioritize requirements, and

how requirements relate to the selection of COTS components.

9.1.1 Requirements Engineering Paradigms

Most requirements engineering methods, and software development methods in general, are Taylorian in nature. Around the turn of this century, Taylor introduced the notion of 'scientific management', in which tasks are recursively decomposed into simpler tasks and each task has one 'best way' to accomplish it. By careful observations and experiments this one best way can be found and formalized into procedures and rules. Scientific management has been successfully applied in many a factory operation. The equivalent in requirements engineering is to interview domain experts and observe end users at work in order to obtain the 'real' user requirements. After this, the experts go to work and implement these requirements. During the latter process there is no further need to interact with the user community. This view of software development is a functional, and rational, one. Its underlying assumption is that there is one objective truth, which merely needs to be discovered during the analysis process.

Though this view has its merits in drawing up requirements in purely *technical* realms, many UoDs of interest involve people as well -- people whose model of the world is incomplete, subjective, irrational, and may conflict with the world view of others. In such cases, the analyst is not a passive outside observer of the UoD. Rather, he actively participates in the *shaping* of the UoD.

It is increasingly being recognized that the Taylorian, functional, approach is not the only, and need not be the most appropriate, approach to the requirements engineering process.

Analysts have a set of assumptions about the nature of the subject of study. Such a set of assumptions is commonly called a 'paradigm'. In our field, these assumptions concern the way in which analysts acquire knowledge (epistemological assumptions) and their view of the social and technical world (ontological assumptions).

The assumptions about knowledge result in an objectivist--subjectivist dimension. If the analyst takes the objectivist point of view, he applies models and methods derived from the natural sciences to arrive at the one and only truth. In the subjectivist position, his principal concern is to understand how the individual creates, modifies and interprets the world he or she is in.

The assumptions about the world result in an order--conflict dimension. The order point of view emphasizes order, stability, integration, and consensus. On the other hand, the conflict view stresses change, conflict, and disintegration.

These two dimensions and their associated extreme positions yield four paradigms for requirements engineering and, more generally, information systems development:

Functionalism (objective--order). In the functionalist paradigm, the developer is the system expert who searches for measurable cause--effect relationships. An empirical organizational reality is believed to exist, independent of the observer. Systems are developed to support rational organizational operation. Their effectiveness and

efficiency can be tested objectively, by tests similar to those used in other engineering disciplines.

Social-relativism (subjective--order). In this paradigm, the analyst operates as a facilitator. Reality is not something immutable 'out there', but is constructed in the human mind. The analyst is a change agent. He seeks to facilitate the learning of all people involved.

Radical-structuralism (objective--conflict). In the radical paradigm the key assumption is that system development intervenes in the conflict between two or more social classes for power, prestige, and resources. Systems are often developed to support the interests of the owners, at the expense of the interests of labor. In order to redress the power balance, this paradigm suggests that the analyst should act as a labor partisan. System requirements should evolve from a cooperation between labor and the analyst. This approach is thought to lead to systems that enhance craftsmanship and working conditions.

Neohumanism (subjective--conflict). The central theme in this paradigm is emancipation. Systems are developed to remove distorting influences and other barriers to rational discourse. The system developer acts as a social therapist in an attempt to draw together, in an open discussion, a diverse group of individuals, including customers, labor, and various levels of management.

Admittedly, these paradigms reflect extreme orientations. In practice, some mixture of assumptions will usually guide the requirements engineering process. Yet it is fair to say that the majority of system development techniques emphasizes the functionalist view.

In the subjectivist--objectivist dimension, it is important to realize that a good deal of subjectivism may be involved in the shaping of the UoD. If we have to develop a system to, say, control a copying machine, we may safely take a functional stand. We may expect such a machine to operate purely rationally. In the analysis process, we list the functions of the machine, its internal signals, conditions, and so on, in order to get a satisfactory picture of the system to be developed. Once these requirements are identified, they can be frozen and some waterfall-like process model can be employed to realize the system.

If, however, our task is to develop a system to support people in doing their job, such as some office automation system, a purely functional view of the world may easily lead to ill-conceived systems. In such cases, end-user participation in the shaping of the UoD is of paramount importance. Through an open dialog with the people concerned, we may encourage the prospective users to influence the system to be developed. Part of the analyst's job in this case is to reconcile the views of the participants in the analysis process. Continuous feedback during the actual construction phases with possibilities for redirection may further enhance the chance of success. It is the future users who are going to work with the system. It is of no avail to confront them with a system that does not satisfy their needs.

Automation transforms organizations, and thus affects the organization's employees. It may raise fears and other emotions with the employees affected. For instance, our library system potentially gives people access to a lot more information than they previously had. Some people though may prefer to have access only to information related to their tasks and responsibilities. During requirements engineering, we have to be conscious about these effects. The pure functionalist paradigm then is of no avail.

A dissatisfied user will try to neglect the system or, at best, express additional requirements immediately. The net result is that the envisaged gain in efficiency or effectiveness is not reaped.

An illuminating and well-documented example of possible effects of following a fairly radical paradigm is given in (Page et al., 1993); see also section 1.4.3. The system concerns the Computer Aided Despatch System for the London Ambulance Service. Though the system would significantly impact the way ambulance crews carried out their jobs, there was little consultation with them. Some of the consequences of this approach were the following (Page et al., 1993, pp 40--41):

- The system allocated the nearest available resource regardless of originating station, so crews often had to operate further and further from their home base. This resulted in them operating in unfamiliar territory with further to go to reach their home station at the end of a shift.
- The new system took away the flexibility crews previously had for the station to decide on which resource to allocate. This inevitably led to problems when a different resource was used to the one that was allocated.
- The lack of voice contact made the whole process more impersonal and exacerbated the 'them and us' situation.

If the conceptual models of the participants differ, we may either look for a compromise, or opt for one of the views expressed. It is impossible to give general guidelines on how to handle such cases. Looking for a compromise can be a tedious affair and may lead to a system that no one is really happy with. Opting for one particular view of the world will make one party happy, but may result in others completely neglecting the system developed. Worse yet, they may decide to develop a competing system.

9.1.2 Requirements Elicitation Techniques

The two main sources of information for the requirements elicitation process are the users and the (application) domain. These sources both presuppose that there exists something 'out there' to start with, and from which requirements can be elicited. In market-driven software development though, this is often not the case, and requirements elicitation in such projects is more like requirements invention or problem-formulation, guided by marketing and sales considerations.

Figure 9.3 lists a number of elicitation techniques, which are elaborated upon below. The figure also tells us that the user is the major source of information in some techniques, while the domain is predominant in others. Furthermore, the figure indicates whether each technique is particularly useful to model the current, as opposed to the anticipated future, situation.

You should generally vacuum a rug in two directions rather than one; likewise, you should use multiple requirements elicitation techniques.

Technique	Main info source		Strong on	
	Domain	User	Current	Future
Interview		X	X	
Delphi technique		X	X	
Brainstorming session		X		X
Task analysis		X	X	
Scenario (use-case) analysis		X	X	X
Ethnography	X		X	
Form analysis	X		X	
Analysis of natural language descriptions	X		X	
Synthesis of reqs from an existing system	X		X	
Domain analysis	X		X	
Use of reference models	X		X	
Business Process Redesign (BPR)	X		X	X
Prototyping		X		X

Figure 9.3 A sample of requirements elicitation techniques

Asking We may simply ask the users what they expect from the system. A presupposition then is that the user is able to bypass his own limitations and prejudices. Asking may take the form of an interview, a brainstorm, or a questionnaire. In an open-ended interview, the user freely talks about his tasks. This is the easiest form of requirements elicitation, but it suffers from all of the drawbacks mentioned before. In a structured interview, the analyst tries to overcome these by leading the user, for example through closed or probing questions.

In discussion sessions with a group of users, we often find that some users are far more articulate than others, and thus have a greater influence on the outcome. The consensus thus reached need not be well-balanced. To overcome this problem, a Delphi technique may be employed. The Delphi technique is an iterative technique in which information is exchanged in a written form until a consensus is reached. For example, participants may write down their requirements, sorted in order of importance. The sets of requirements thus obtained are distributed to all participants,

who reflect on them to obtain a revised set of requirements. This procedure is repeated several times until sufficient consensus is reached.

For consumer products, such as word processing packages, antivirus software or software for your personal administration, users often have the option to give feedback, raise questions, report bugs, and the like, electronically. This type of information is also regularly gathered and stored by sales and marketing people in the course of their contacts with customers. These logs can be mined and in this way provide a valuable source of information when looking for requirements for the next release of that software.

Task analysis Employees working in some domain perform a number of tasks, such as handling requests to borrow a book, cataloging new books, ordering books, etc. Higher-level tasks may be decomposed into subtasks. For example, the task 'handle request to borrow a book' may lead to the following subtasks:

- check member identification,
- check for limit on the number of books that may be borrowed,
- register book as being borrowed by the library member,
- issue a slip indicating the due back date.

Task analysis is a technique to obtain a hierarchy of tasks and subtasks to be carried out by people working in the domain. Any of the other techniques discussed may be used to get the necessary information to draw this hierarchy. There are no clear-cut rules as to when to stop decomposing tasks. A major heuristic is that at some point users tend to 'refuse' to decompose tasks any further. For instance, when being asked how the member identification is checked, the library employee may say 'Well, I simply check his id.' At this point, further decomposition is meaningless.

Task analysis is often applied at the stage when (details about) the human--computer interaction component are being decided upon. This underestimates its potency as a general requirements elicitation technique. It also gives the (wrong) impression that users are only concerned with the 'look and feel' of the interface.

Scenario-based analysis Instead of looking for generic plans as in interviews or task analysis, the analyst may study **instances** of tasks. A scenario is a story which tells us how a specific task instance is executed. The scenario can be real or artificial. An example of a real scenario is that the analyst observes how a library employee handles an actual user request. We may ask the library employee to verbalize what he is doing and make an audio or video recording thereof. This **think aloud** method is a fairly unobtrusive technique to study people at work. It is often used to assess prototypes or existing information systems.

Alternatively, we may construct artificial scenarios and discuss these with the user. As a first shot, we may for example draw up the following scenario for returning a book:

1. The due back date for the book is checked. If the book is overdue, the member is asked to pay the appropriate fine.
2. The book is recorded as again being eligible for checking-out.
3. The book is put back in its proper place.

When this scenario is discussed with the library employee, a number of related issues may crop up, either through probing questions from the analyst, or because the user contrasts the scenario with daily practice. Example questions that could be raised include such things as:

- What happens when the person returning the book is not a registered member of the library?
- What happens when the book returned is damaged?
- What happens if the member returning this book has other books that are overdue or an outstanding reservation for another book?

In essence, this type of story-telling provides the user with an artificial mock-up version of the software eventually to be delivered. It serves as a paper-based prototype to gain a better understanding of the requirements. If tied to a UML-type of modeling, scenario-based analysis is often called **use-case analysis**; see section 10.3.6. Scenarios and use cases are the elicitation methods most often used.

Scenario-based analysis is often done in a somewhat haphazard way. In that case, there is no way of telling whether enough scenarios have been drawn up and a sufficiently accurate and complete picture of the requirements is obtained. Writing good scenarios is by no means easy. Though it may look trivial to 'just record user episodes', a fair mount of domain expertise is needed to get a good and reliable set of scenarios.

Scenarios can be looked at from different perspectives. In the above example scenario for returning a book, the scenario lists a series of actions or events that together make up some episode. The focus then is on the process aspect, showing how the system proceeds through successive states. Alternatively, the same scenario may be looked at from a user perspective: how will the user interact with the system, what functionalities will she be offered? Yet another perspective is that the scenario leads to discussions about alternatives from which a certain choice has to be made, as in the questions that the example scenario above raised.

Ethnography A major disadvantage of eliciting requirements through, for example, interviews is that the analyst imposes his view of how the world is ordered onto the user. Such methods may fail if the analyst and user do not share a category system. The analyst may, for example, ask the following:

'If a member wants to borrow a book while he or she still has an outstanding fine, will you:

a) Refuse the request, or

b) Handle the request anyway'

This binary choice need not map actual practice. The library employee may, for example, grant the request provided part of the outstanding fine is settled or if he knows the member to be trustworthy.

Thinking aloud protocols are based on the idea that users have well-defined goals and subgoals, and that they traverse such goal trees in a neat top-down manner. People however often do not have preconceived plans, but rather proceed in somewhat opportunistic ways.

A disadvantage of task analysis is that it considers individual tasks of individual persons, without taking into account the social and organizational environment in which these tasks are executed.

Ethnographic methods are claimed not to have such shortcomings. In ethnography, groups of people are studied in their natural settings. It is well-known from sociology, where for example Polynesian tribes are studied by living with them for an extended period of time. Likewise, user requirements can be studied by participating in their daily work for a period of time, for example by becoming a library employee. The analyst becomes an apprentice, recognizing that the future users of the system are the real experts in their work. Ethnographic methods are more likely to uncover tacit knowledge than most other elicitation techniques.

Form analysis A lot of information about the domain being modeled can often be found in various forms being used. For example, to request some conference proceedings from another library, the user might have to fill in a form such as given in figure 9.4.

Proceedings Request Form	
Member name	.....
Member address	.....
	.....
Title	.....
Series no	.....
Editor	.....
Place	.....
Publisher	.....
Year	.....
Signature	.....

Figure 9.4 A sample form

Forms provide us with information about the data objects of the domain, their properties, and their interrelations. They are particularly useful as an input to modeling the data aspect of the system; see also section 10.1.1.

Library users often have incomplete knowledge of the information sources they are interested in. For example, someone might be looking for the proceedings of the *International Conference on Software Engineering* that took place in Berlin. Only if the various entries from the above form are used as entities in the underlying data model, can such a query be answered easily. In this case, the form directly points at a useful requirement which might otherwise go unnoticed.

Natural language descriptions Like forms, natural language descriptions provide a lot of useful information about the domain to be modeled. The operating instructions for library employees might for instance contain a paragraph like the one given in figure 9.5. This text gives us such information as:

- There are (at least) two accounts that orders can be charged to;
- There is a list of staff members authorized to sign off such requests;
- There is the possibility of ordering multiple copies of titles, such as this book on *Software Engineering*, on behalf of students.

Title acquisition

Before a request to acquire a title can be complied with, form B has to be filled in completely. A request can not be handled if it is not signed by an authorized staff member or the account to be charged ('Student' or 'Staff') is not indicated. A request is not to be granted if the title requested is already present in the title catalog, unless it is marked 'Stolen' or 'Lost', or the account is 'Student'.

Figure 9.5 A sample instruction for library employees

Often, natural language descriptions (and forms) provide the analyst with background information to be used in conjunction with other elicitation techniques such as interviews. Natural language descriptions in particular tend to assume a lot of tacit knowledge by the reader. For example, if form B contains an ISBN, this saves the library employee some work, but the request will probably still be handled if this information is not provided. A practical problem with natural language descriptions is that they are often not kept up-to-date. Like software documentation, their validity tends to deteriorate with time.

Natural language descriptions are often taken as a starting point in object-oriented analysis techniques. This is further discussed in section 12.3.

Derivation from an existing system Starting from an existing system, for instance a similar system in some other organization or a description in a text book, we may formulate the requirements of the new system. Obviously, we have to be careful and take the peculiar circumstances of the present situation into account.

Rather than looking at one particular system, we may also study a number of systems in some application domain. This meta-requirements analysis process is known as **domain analysis**. Its goal generally is to identify reusable components, concepts, structures, and the like. It is dangerous to look for reusable requirements in immature domains. Requirements may then be reused simply because they are available, not because they fit the situation at hand. They become 'dead wood'. In the context of requirements analysis, domain analysis can be viewed as a technique for deriving a 'reference' model for systems within a given domain. Such a reference model provides a skeleton (architecture) that can be augmented and adapted to fit the specific situation at hand.

Domain analysis is further discussed in chapter ??, in the context of software reuse and software product line development.

Business Process Redesign (BPR). In many software development projects, the people involved jump to conclusions rather quickly: automation is the answer. Even worse, their conclusion might be that automating the current situation is the answer. In Business Process Redesign (or Business Process Reengineering), a rather different strategy is followed. It is an organizational activity to radically redesign business processes to achieve competitive breakthroughs in, e.g. quality, cost, or user satisfaction. In BPR, we depart completely from the existing ways of doing things. In BPR, the following steps are distinguished:

1. Identify processes for innovation. Two major approaches for doing so are the exhaustive and high-impact approach. In the exhaustive approach, an attempt is made to identify all processes, which are then prioritized for their redesign urgency. The high-impact approach attempts to identify the most important processes only, or the ones that conflict with the business vision.

2. Identify change levers. In this step, opportunities facilitating process improvement are identified. Three types of lever can be recognized: organizational enablers (such as empowering teams), human resource enablers (such as task enrichment) and information technology enablers.

3. Develop process visions. For redesign to be successful, the organization needs to know which goals it wants to reach. This is described in the process vision. The main components of a process vision are: process objectives (measurable targets of the future performance of the system), process attributes (qualitative and descriptive properties of the future process), critical success factors and constraints (organizational, cultural, and technological).

4. Understand the existing process. This includes documenting the existing process, measuring it, and identifying problematic aspects. It allows us to assess the health of the existing process, and brings problems to the surface.

5. Design and prototype the new process. This is the final step. Prototyping makes it possible to try out new structures, thereby reducing the risk of failure.

BPR is not really a requirements elicitation technique proper. It is mentioned here because it emphasizes an essential issue to be addressed during the requirements engineering phase. Business processes should not be driven by information technology. Rather, information technology should enable them. Though a complete BPR effort is not necessary or feasible in many situations, rethinking the existing processes and procedures is a step which is all too often thoughtlessly skipped in software development projects.

As an example, consider our library automation project once again. Careful inspection of the current situation might reveal that things aren't all that bad. However, the impression is that the number of requests that could not be granted has steadily risen in the past years. This is perceived to be the main cause of the increasing number of dissatisfied users. Since service to its customers has high priority, one of the objectives is to decrease the number of requests that cannot be satisfied by 50% within two years. For this to be possible, the library should be allowed to spend the available budget at its own discretion, rather than being triggered by signals from researchers only (this sounds radical, doesn't it). It is therefore decided to augment the existing automated system with modules to keep track of both successful and unsuccessful requests. Based on the insights gained from this measurement process during a period of three months, a decision will be taken as to how large a percentage of the annual budget will be reallocated.

Prototyping Given the fact that it is difficult, if not impossible, to build the right system from the start, we may decide to use prototypes. Starting from a first set of requirements, a prototype of the system is constructed. This prototype is used for experiments, which lead to new requirements and more insight into the possible uses of the system. In one or more ensuing steps, a more definite set of requirements is developed. Prototyping is discussed in section 3.2.1. Other agile processes follow a similar strategy in which requirements are quickly translated into a running system to be assessed by its users.

Of these requirements elicitation techniques, asking is the least certain strategy, while prototyping is the least uncertain. Besides the experience of both users and analysts, the uncertainty of the process is also influenced by the stability of the environment, the complexity of the product to be developed and the familiarity with the problem area in question. We may try to estimate the impact of those factors on the vulnerability of the resulting requirements specification, and then decide on a certain primary method for requirements elicitation based on this estimate.

For a well-understood problem, with very experienced analysts, interviewing the prospective users may suffice. However, if it concerns an advanced and ill-understood problem from within a rapidly changing environment and the analysts have little or no experience in the domain in question, it seems wise to follow an agile process.

Requirements uncertainty is not the only problem project managers have to cope with, and a different process is not the only solution they opt for. Political aspects (such as hidden agendas and conflicts between stakeholders) are often seen as larger risks than mere requirements uncertainty (Moynihan, 2000). Of course, these are related. In both cases chances are high that requirements will change. Project managers often follow a formal route to handle disagreements between stakeholders, and let the customers sign off the requirements document. Whether this is the answer in the long run is questionable, though.

As the uncertainty decreases, the beneficial effects of user participation in requirements engineering diminish. If the uncertainty increases, however, greater user participation does have a positive effect on the quality of requirements engineering.

It is generally wise to have multiple customer--developer links in a software development project, and during requirements engineering in particular. Keil and Carmel (1995) studied the relation between project success and the number and type of such customer--developer links. The authors observed a strong correlation between the number of links and project success: more links implied more successful projects. The relative contribution to project success diminishes as the number of links grows; there is no need to have more than, say, half a dozen links. A further interesting observation from this study is that links with *direct* users have more impact on project success than links with *indirect* users such as user representatives or sales people. Finally, it was noted that customer-driven development projects tend to use and prefer different types of link to market-driven development projects. For example, the favorite link for custom development -- facilitated teams -- was not used by package developers, while the favorite link for package developers -- support lines -- was seldom used for custom projects.

We should be very careful in our assessment of which requirements elicitation technique to choose. It is all too common to be too optimistic about our ability to properly assess software requirements.

As an example, consider the following anecdote from a Dutch newspaper. A firm in the business of farm automation had developed a system in which microchips were put in cows' ears. Subsequently, each individual cow could be tracked: food and water supply was regulated and adjusted, the amount and quality of the milk automatically recorded and analyzed, etc. Quite naturally, this same technique was next successfully applied to pigs. Thereafter, it was tried on goats. A million-dollar, fully automated goat farm was built. But alas, things did not work out that well for goats. Contrary to cows and pigs, goats eat everything, including their companions' chips.

9.1.3 Goals and Viewpoints

In this section we discuss two ways to structure a set of requirements. One way to do so is in a hierarchical structure: higher-level requirements are decomposed into lower-level ones. The high-level requirements are often termed goals. The other structuring method links requirements to specific stakeholders. For example, management may have a set of requirements, and the end-users may have (another) set of requirements. These different sets of requirements are called viewpoints. In both cases, elicitation and structuring go hand in hand.

For example, one of the requirements elicited for our library system could be that the system should allow users to search the database for a particular book. By asking ourselves or the stakeholders *why* this requirement is needed, a higher level requirement is detected, viz. the necessity for having search facilities. Again asking why, a high-level goal of serving the customers is arrived at. Going the other way, by asking *how* the library system may help serve the customers, a requirement to learn about user preferences and use this knowledge while interacting with the user, might emerge. In this way, by asking *why* and *how* questions, a hierarchical structure of goals and requirements develops. Figure 9.6 contains an example of such a hierarchical structure.

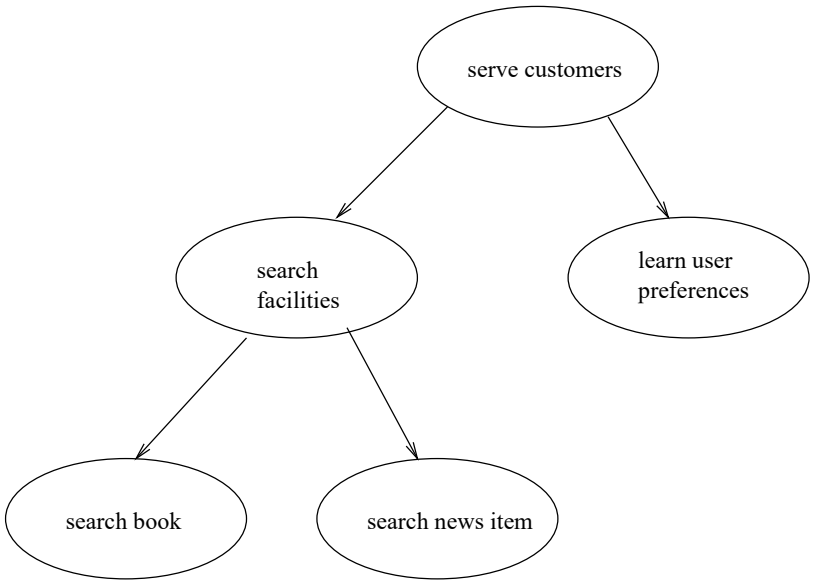

Figure 9.6 Hierarchy of requirements

Figure 9.6 depicts a refinement structure, in which each requirement is refined (decomposed) into a set of subrequirements that together satisfy the parent requirement. The subrequirements are AND-related: 'search book' and 'search news item' together make up the 'search facilities' requirement.

We may also include other types of relationships. For instance, we may have certain options for a particular requirement, and to that end have OR-relations next to AND-relations. We may also include other types of link. If we have a requirement to impose fines on customers who return items late, we may conceive this as conflicting with our goal of serving customers, and connect these two requirements by a link of type 'conflicts with'.

This so-called **goal-driven requirements engineering** results in a graph connecting high-level goals to lower-level requirements. This graph can be reasoned about, e.g. to validate that certain goals are indeed reached, or to detect conflicts (Lamsweerde, 2001).

It is often useful to collect and organize requirements from different perspectives, or **viewpoints**. Different stakeholders may have different sets of requirements. Different quality concerns may also lead to different sets of requirements, leading for instance to a security viewpoint. The latter type of perspective is usually dealt with during software architecture design, and will be discussed in chapter 11. The techniques discussed in section 9.3 implicitly denote different viewpoints as well, such as a data viewpoint in the entity-relationship models. Here, we focus on different viewpoints caused by different stakeholders. These different viewpoints may be in conflict, and these conflicts need to be recognized and dealt with during requirements engineering.

The Computer Aided Dispatch System for The London Ambulance Service again provides a clear case of conflicting viewpoints: management wants an effective system, crew members want to get home within a reasonable time after their shift has ended (see also sections 1.4.3 and 9.1.1). For our library system, conflicts between stakeholders may likewise occur.

Consider, for example, the following issue which may crop up during the requirements elicitation phase for our library system. The system has to offer certain features to register and handle fines. An item not returned in time incurs a fine of, say, $0.25 per day. John, one of the library employees involved in the specification process, takes the following position (denoted 'Pos A' in figure 9.7): members should be warned about outstanding fines at the earliest possible moment. His argument ('Arg A') is that service is degraded if a member cannot borrow an item because some other member has not returned that item on time. Mary, the library manager, takes a rather different position ('Pos B'): members should *not* be warned about outstanding fines until the due-back date has expired one month. Her argument ('Arg B') is that fines are a most welcome addition to the library budget, which is under severe pressure because of the continuing price increase of journal subscriptions.

This situation is graphically depicted in the graph in figure 9.7. The graph contains nodes of types 'issue', 'position' and 'argument', and directed links of type 'response-to',

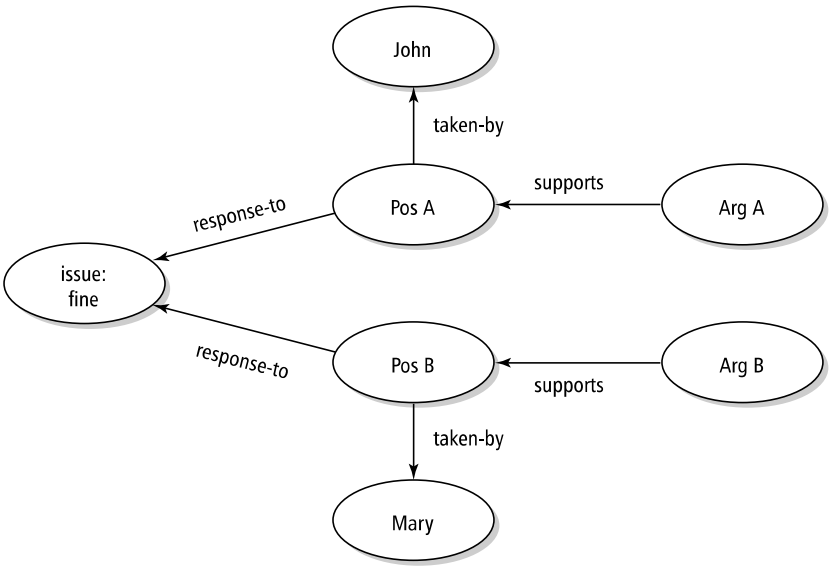

Figure 9.7 Graph representation of conflicting viewpoints

'taken-by' and 'supports'. Capturing this type of information in an automated system offers possibilities to store, trace and manipulate the very diverse types of information being gathered during the requirements engineering phase. An early system along these lines is gIBIS, a system designed to capture early design decisions.

Two viewpoints in particular are important during requirements engineering: the business viewpoint and the personal viewpoint. The business viewpoint is usually propagated by management stakeholders, while the personal viewpoint is usually propagated by end users. However, end users tend to also ascribe to business requirements, at least at an early stage. For instance, when John is asked whether fines are a welcome addition to the subsidy the library gets from the government, a likely answer is 'yes'. This requirement is viewed as a requirement of the business, not a personal requirement of John. Only when he is confronted with the consequences, will he realize that this is after all not what he wants. And a request to change the system will follow.

9.1.4 Prioritizing Requirements

Our task is not to provide every button and pull-down menu enhancement that our customers ask for, but to invent a completely new way of working -- one that will thrill

and amaze them.
(Robertson, 2002)

In most cases, not all requirements can be realized, so we have to make a selection. In section 3.2.3 we mentioned a very simple form of requirements prioritization called *triage*. A variant often used is known as MoSCoW (the o's are just there to be able to pronounce the word). Usinig MoSCoW, we distinguish four types of requirement:

- **Must haves**: these are the top priority requirements, the ones that definitely have to be realized in order to make the system acceptable to the customer.
- **Should haves**: these requirements are not strictly mandatory, but they are highly desirable.
- **Could haves**: if time allows, these requirements will be realized as well. In practice, they usually won't.
- **Won't haves**: these requirements will not be realized in the present version. They are recorded though. They will be considered again for a next version of the system.

The MoSCoW scheme assumes that requirements can be ordered along a single axis, and that realizing more requirements yields more satisfied customers. Reality often is more complex. In the **Kano model** (Kano, 1993), user preferences are classified into five categories, as listed in figure 9.8. The way customers value the Attractive, Must-be and One-dimensional categories of requirements is depicted in figure 9.9. This figure shows that offering attractive, so called killer features, is what will really excite your customers. The above quote from Robertson (2002) points in the same direction: amaze your customer by giving him something he never even dreamt of.

In market-driven software development, the product often has a series of releases. The list of requirements for such products is usually derived from sales information, user logs from earlier versions of the system, and other sources of indirect information. One then has to decide which requirements to include in the current version, and which ones to postpone to a next one. Business case analysis, return on investment estimations, and similar economics-driven argumentations then are used to set priorities. This priority setting is to be repeated for each version, since user preferences may change, the market reacts, and the like.

Finally, the prioritization of requirements is related to the notion of *scoping* in software product lines. If we want to develop a series of similar library systems, we have to delimit the domain we intend to handle. A smaller domain, say only scientific libraries, is easier to realize, but has a smaller market. A set of products covering a larger domain is more difficult to realize, yet has the promise of larger sales and profits. Scoping is further discussed in chapter ??, in the context of our discussion of software product lines.

Attractive	The customer will be more satisfied if these requirements are met, but not les satisfied if they are not. For example, an automatic alert if new books of a beloved author arrive.
Must-be	The customer will be dissatisfied if these requirements are not met, but his satisfaction will never raise above neutral. An example is the ability to search the library catalog.
One-dimensional	For requirements of this category, satisfaction is proportional to how many of these requirements are being met. Alternative ways to search the library catalog could fall into this category.
Indifferent	The customer does not really care about these requirements. For example, the customer might not care whether different categories of library items are displayed in a different color on the screen.
Reverse	The customer's judgement of the requirements is the opposite of what the analyst thought a priori. For example, the analyst may have thought the library customer would want the system to remember her search patterns so as to be able to serve her better next time, while the customer wants to start afresh each time.
Questionable	The customer's preferences are not clear. She both seems to like and dislike a certain feature.

Figure 9.8 Kano's requirements categories

9.1.5 COTS selection

Up till now, we dealt with a situation where the customer phrases her requirements, after which a system that satisfies these requirements is developed. With Commercial Off The Shelf (COTS) software, the customer has to choose from what is available. In practice, the situation is not always that clear cut, and the COTS system may be extended or adapted to suit the customer's needs. For our discussion, we assume it is a pure selection process.

COTS selection is an iterative process comprising the following steps:

1. Define requirements. As in ordinary requirements elicitation processes, a list of requirements for the product is derived. Any of the elicitation techniques discussed above may be used in this process.

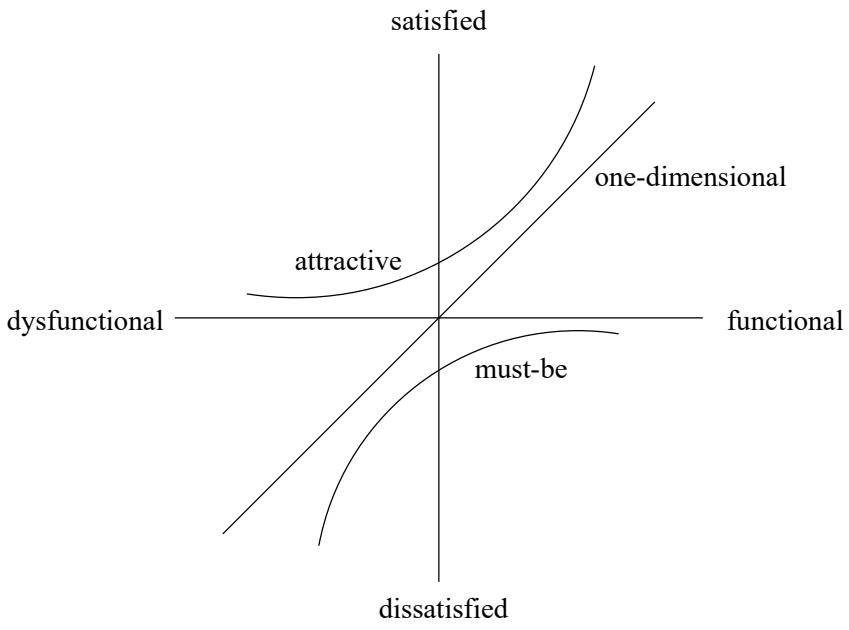

Figure 9.9 The Kano diagram

2. Select components. A set of components that can possibly handle the requirements posed is determined. This selection process may involve market research, internet browsing, and a variety of other techniques.

3. Rank the components. The components are ranked in the order in which they satisfy the requirements.

4. Select the most appropriate component, or iterate.

Often, the set of components and requirements are too large to make a complete analysis and ranking in one step feasible. An iterative process is then followed, whereby the most important requirements are used to make a first selection from the set of available components. In a next step, a larger list of requirements is assessed against a smaller set of components. And so on.

There are different ways to rank components. A straightforward method is the Weighted Scoring Method (WSM). Each requirement is given a weight, and the alternatives are given a score for each requirement, say on a scale from 1 to 5. In table 9.1, three components labeled A, B, and C are scored on three criteria: performance, supplier reputation, and functionality. In the example, components B and C score highest, and these might next be scrutinized further.

Table 9.1 WSM example

Criteria	Weight	A	B	C
Performance	2	1	3	5
Supplier	1	2	2	5
Functionality	3	4	5	1
Total		16	23	18

A major drawback of WSM is that every criterion can be compensated for by any other criterion. In the example from table 9.1, component C makes it to the next round even though it scores very low on functionality. More complex ranking schemes, such as the Analytic Hierarchy Process (AHP) overcome this drawback ((Saaty, 1990)).

9.2 Requirements Documentation and Management

The end-product of the requirements engineering phase in a document-driven development project is a requirements specification. The requirements specification is an a posteriori reconstruction of the results of this analysis phase. Its purpose is to communicate these results to others. It serves as an anchor point against which subsequent steps can be justified.

The requirements specification is the starting point for the next phase: design. Consequently, a very precise, even mathematical description is preferable. On the other hand, the specification must also be understandable to the user. This often means a readable document, using natural language and pictures. In practice, one has to look for a compromise. Alternatively, the requirements specification may be presented in different, but consistent, forms to the different audiences involved.

Besides readability and understandability, various other requirements for this document can be stated (IEEE830, 1993):

- A requirements specification should be *correct*. There is no procedure to guarantee correctness. The requirements specification should be validated against other (superior) documents and the actual needs of the users to assess its correctness.

- A requirements specification should be *unambiguous*, both to those who create it and to those who use it. We must be able to uniquely interpret requirements. Because of its very nature, this is difficult to realize in a natural language.

- A requirements specification should be *complete*. All significant matters relating to functionality, performance, constraints, and the like, should be documented. The responses to both correct and incorrect input should be specified; phrases

like 'to be determined' are particularly insidious. Unfortunately, it is not always feasible to complete the specification at an early stage. If certain requirements can only be made specific at a later stage, the requirements specification should at least document the ultimate point in time at which this should have happened.

- A requirements specification should be (internally) *consistent*, i.e. different parts of it should not be in conflict with each other. Conflicting requirements can be both logical and temporal. Using different terms for one and the same object may also lead to conflicts.

- Requirements should be ranked for *importance* or *stability*. Typically, some requirements are more important than others. In some cases, a simple ranking scheme like 'essential', 'worthwhile', and 'optional' will suffice; in other cases, a more sophisticated classification scheme may be needed (see also section 9.1.4). We may indicate the stability of requirements by indicating the likelihood, or the expected number, of changes. Through the explicit incorporation of this type of information in the requirements document, users are stimulated to give more consideration to each requirement. It also gives developers the opportunity to better direct their attention.

- A requirements specification should be *verifiable*. This means that there must be a finite process to determine whether or not the requirements have been met. Phrases like 'the system should be user-friendly' are not verifiable. Likewise, the use of quantities that cannot be measured, as in 'the system's response time should usually be less than two seconds', should be avoided. A requirement like 'for requests of type X, the system's response time is less than two seconds in 80% of cases, with a maximum machine load of Y', is verifiable.

- A requirements specification should be *modifiable*. Software models part of reality. Therefore it changes. The corresponding requirements specification has to evolve together with the reality being modeled. Thus, the document must be organized in such a way that changes can be accommodated readily (a tabular or database format, for example). Redundancy must be prevented as much as possible, for otherwise there is the danger that changes lead to inconsistencies.

- A requirements specification should be *traceable*. The origin and rationale of each and every requirement must be traceable. A clear and consistent numbering scheme makes it possible that other documents can uniquely refer to parts of the requirements specification.

As a guideline for the contents of a requirements specification we will follow IEEE Standard 830. This standard does not give a rigid form for the requirements specification. In our opinion, the precise ordering and contents of the elements of this

document also is less essential. The important point is to choose a structure which adheres to the above constraints. In IEEE Standard 830, a global structure such as depicted in figure 9.10, is used.

1. *Introduction*
   - 1.1 Purpose
   - 1.2 Scope
   - 1.3 Definitions, acronyms and abbreviations
   - 1.4 References
   - 1.5 Overview
2. *Overall description*
   - 2.1 Product perspective
   - 2.2 Product functions
   - 2.3 User characteristics
   - 2.4 Constraints
   - 2.5 Assumptions and dependencies
   - 2.6 Requirements subsets
3. *Specific requirements*

Figure 9.10 Global structure of the requirements specification (*Source: IEEE Recommended Practice for Software Requirements Specifications*, IEEE Std 830, 1993. *Reproduced by permission of IEEE.*)

For any nontrivial system, the detailed requirements will constitute by far the largest part of the requirements document. It is therefore helpful to somehow categorize these detailed requirements. This can be done along different dimensions, such as:

- **Mode**. Systems may behave differently depending on the mode of operation, such as training or operational. For example, performance or interface requirements may differ between modes.
- **User class**. Different functionality may be offered to different classes of users, such as library members and library personnel.
- **Objects**. Requirements may be classified according to the objects (real-world entities) concerned. This classification scheme is a natural one when used in conjunction with an object-oriented analysis technique (see section 12.3).
- **Response**. Some systems are best described by placing together functions in support of the generation of a response, for example functions associated with catalog queries or library member status information.

- **Functional hierarchy**. When no other classification fits, some functional hierarchy, for example organized by common inputs, may be used.

As an example, figure 9.11 gives a refinement
of the section on specific requirements along the dimension of user classes,

3. *Specific requirements*
   - 3.1 External interface requirements
     - 3.1.1 User interfaces
     - 3.1.2 Hardware interfaces
     - 3.1.3 Software interfaces
     - 3.1.4 Communications interfaces
   - 3.2 Functional requirements
     - 3.2.1 User class 1
       - 3.2.1.1 Functional requirement 1.1
       - 3.2.1.2 Functional requirement 1.2
       - . . .
     - 3.2.2 User class 2
       - . . .
   - 3.3 Performance requirements
   - 3.4 Design constraints
   - 3.5 Software system attributes
   - 3.6 Other requirements

Figure 9.11 Prototype outline of the section on Specific Requirements (*Source: IEEE Recommended Practice for Software Requirements Specifications*, IEEE Std 830-1993. *Reproduced by permission of IEEE.*)

A further clarification of the various components is given in Appendix ??. As an example, figure 9.12 contains (part of) a possible requirements specification for the library example mentioned earlier, following the IEEE guidelines.

Start of figure 9.12

1. *Introduction.*

   1.1 *Purpose.* This document states the requirements of an automated library system for a medium-sized library of a research institute. The requirements stated serve as a basis for the acceptance procedure of this system. The document is also intended as a starting point for the design phase.

1.2 *Scope*. The intended product automates the library functions described in DOC1. Its purpose is to provide a more effective service to the library users, in particular through the online search facilities offered. More details of the performance requirements are given in section 3.3 of this document. Once this system is installed, the incorporation of new titles will go from an average of 15 minutes down to an average of 5 minutes.

1.3 *Definitions, acronyms and abbreviations*. Library member: . . . , Library personnel: . . . , User: The term user may refer to both library members and library personnel, and is used to denote either class of users. Title catalog: . . . , PICA: . . . , etc.

1.4 *References*. DOC1: . . . , DOC2: . . . , etc.

1.5 *Overview*. Section 2 of this document gives a general overview of the system. Section 3 gives more specific requirements for functions offered. These functions are categorized according to the class of users they support: (external) members of the library and library personnel, respectively.

2. *Overall description*.

2.1 *Product perspective*. The already installed database system X will be used to store the various catalogs as well as the library member administration. There are no interfaces to other systems. The system will be realized on the Y configuration. The maximum external storage capacity for the catalogs of the system is 1500 MB. Library personnel will use a barcode reader to enter member, book and journal identifications. The interface protocol to the barcode reader is described in DOC4.

2.2 *Product functions*. The system provides two types of function:

- Functions by which users may search the catalogs of books and journal articles. A list of these functions is given in DOC1. A more detailed description is given in section 3.2.1.
- Functions by which library personnel may update the administration of borrowed titles and the system's catalogs; see section 3.2.2.

The user of the system selects one of the functions offered through the main menu (section 3.2.1.1 and 3.2.2.1).

2.3 *User characteristics*. The library members are incidental users of the system and have little knowledge of automated systems of this kind. The system therefore has to be self-instructing. Specific requirements are formulated in sections 3.1.1 and 3.3. The library personnel will be trained in the use of the system; see section 3.1.1.

2.4 *Constraints*. Library members may only search the catalogs of books and journal articles; they are not allowed to update a catalog or the user administration. The latter functionality is to be offered through a dedicated, password-protected interface only.

2.5 *Assumptions and dependencies*. . . .

2.6 *Requirements subsets*. . . .

3. *Specific requirements*.

3.1 *External interface requirements*.

3.1.1 *User interfaces*. The screen formats for the different features are specified in Appendix A. Appendix B lists the mapping of commands to function keys. The user can get online help at any point by giving the appropriate command. Appendix C contains a list of typical usage scenarios. These usage scenarios will be used as acceptance criteria: 80% of the users must be able to go through them within ten minutes. An instruction session for library personnel should take at most two hours.

3.1.2 *Hardware interfaces*. The user interface is screen-oriented. The system uses up to ten function keys.

3.1.3 *Software interfaces*. The interface with database system X is described in DOC2.

3.1.4 *Communications interfaces*. Not applicable.

3.2 *Functional requirements*.

3.2.1 *Library member functions*.

3.2.1.1 *Select member feature*. The user selects one of the options from the main menu. Subsequent actions are described in sections 3.2.1.2 and 3.2.1.3.

At any point, the user has an option to return to the main menu (see Appendix B).

3.2.1.2 *Search book catalog*. Given (part of) a book title or author name, the user may search the book catalog for titles that match the input given. The user is offered a screen with two fill-in-the-blank areas (one for the title and one for the author), one of which is to be filled in.

Input. The input may contain both upper and lower case letters. Special symbols allowed are listed in DOC1. Any other glyphs entered are discarded and are not shown on the screen. The input is considered complete when the processing command is issued.

Processing. All lower case letters are turned into upper case letters. The string thus obtained is used when querying the database. A database entry matches the title string given if the transformed input is a substring of the title field of the entry. The same holds for the author field if (part of) an author name is input.

Output. A list of titles that match the input is displayed. Up to four titles are shown on the screen. The user may traverse the list of titles found using the screen scrolling commands provided. A warning is issued if no title matches the input given.

3.2.1.3 *Search article catalog*. . . .

3.2.2 *Library personnel functions*.

3.2.2.1 *Select personnel feature*. Through a dedicated, password-protected interface, library personnel are offered an extended main menu, listing the options available to all users, as well as the options available to library personnel only. The latter are described in sections 3.2.2.2 and 3.2.2.3.

3.2.2.2 *Borrow title*. . . .

3.2.2.3 *Modify catalog*. . . .

3.3 *Performance requirements*. The system will initially support 32 concurrent access points. Its maximum capacity is 128 concurrent access points.

The present database holds 25 000 book titles and 500 journal subscriptions. The storage capacity needed for these data is 300 MB. On average 1000 books and 2000 journal issues enter the library per year. The average journal issue has six articles. This requires a storage capacity of 15 MB per year.

The system must be able to serve 20 users simultaneously. With this maximum load and a database size of 450 MB, user queries as listed in sections 3.2.1 and 3.2.2 must be answered within five seconds in 80% of the cases.

3.4 *Design constraints*.

3.4.1 *Standards compliance*. Title descriptions must be stored in PICA-format. This format is described in DOC3.

3.4.2 *Hardware limitations*. See section 2.1.

3.5 *Software system attributes*.

3.5.1 *Availability*. During normal office hours (9 am--5 pm) the system must be available 95% of the time. A backup of the system is made every day at 5 pm.

3.5.2 *Security*. The functions described in section 3.2.2 are restricted to library employees and password-protected. . . .

3.5.3 *Maintainability*. . . .

3.6 *Other requirements*. . . .

The IEEE framework for the requirements specification is especially appropriate in document-driven models for the software development process: the waterfall model

Figure 9.12 Partly worked-out requirements specification for the library example

End figure 9.12

and its variants. When a prototyping technique is used to determine the user interface, the IEEE framework can be used to describe the outcome of that prototyping process. The framework assumes a model in which the result of the requirements engineering process is unambiguous and complete. Though it is stated that requirements should be ranked for importance, and requirements that may be delayed until future versions may be included as subsets, this does not imply that a layered view of the system can be readily derived from a requirements document drawn up this way.

Irrespective of the format chosen for representing requirements, the success of a product strongly depends upon the degree to which the desired system is properly described during the requirements engineering phase. Small slips in the requirements specification may necessitate large changes in the final software. Software is not continuous, as we noted earlier.

The importance of a solid requirements specification cannot be stressed often enough. In some cases, up to 95% of the code of large systems has had to be rewritten in order to adhere to the ultimate user requirements.

9.2.1 Requirements Management

A fundamental problem with the IEEE framework discussed above is that it describes the end product only. Before this final stage is reached, the 'current' set of requirements is in a constant state of flux. And even after the requirements phase is ended, requirements will change and new requirements will be put forth. The latter phenomenon is known as **requirements creep**, and is the cause for many run-away projects.

Changes in requirements cannot be circumvented. In many cases, it is not wise to aim for an early freeze of the requirements either. In general, the preferred situation is as depicted in figure 9.13: in the course of time, the set of requirements becomes more and more stable.

Obviously, this evolving set of requirements has to be managed. Requirements management involves three activities:

- requirements identification,
- requirements change management, and
- requirements traceability.

Each requirement has to have a unique identification. The simplest form is to just number them. If there is a hierarchical organization, as in a goal-hierarchy, such can be reflected in the numbering scheme. Since requirements are often changed and updated, it is expedient to include versioning information as well. Finally, we may

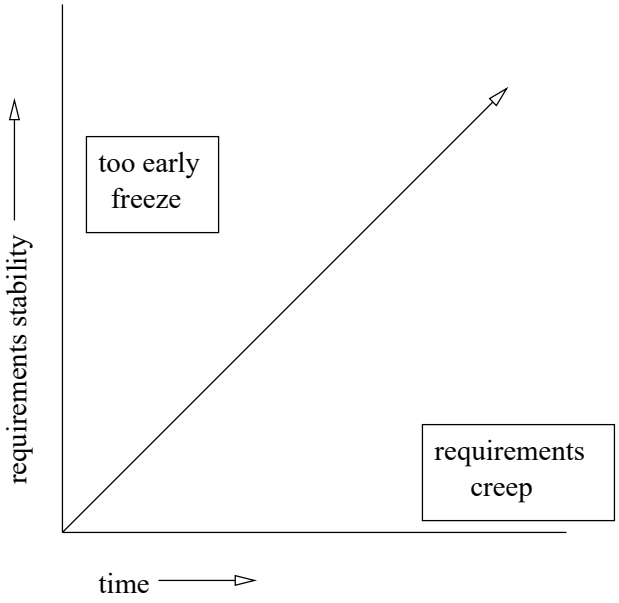

Figure 9.13 Requirements stability over time

add some attributes to each requirement, such as its status, priority, main stakeholder involved, and the like. Requirements engineering tools usually have means to store requirements in a structured repository.

Changes to requirements have to be properly managed. By viewing each requirement as a configuration item, the rules and procedures from configuration management (chapter 4) can be applied.

We may connect requirements to solution elements such as design elements or even software components that realize those requirements. In this way, we establish **traceability** from requirements to code and vice versa. This allows us to trace where requirements are realized (forward traceability), and why certain solutions are chosen (backward traceability). Traceability information is important in all development phases. It can be used to answer a variety of questions, such as:

- where is this requirement implemented?
- do we need this requirement?
- are all requirements linked to solution elements?
- which requirement does this test case cover?
- what is the impact of this requirement?

– do we need this design element (piece of code)?

This way of making the relation between requirements and solutions explicit is closely related to **design space analysis**. In design space analysis, the aim is to explicitly model all possible combinations of requirements and solutions. Design space analysis originated in the field of human-computer interaction. A well-known notation for Design Space Analysis is known as QOC (Questions, Options and Criteria). Questions correspond to requirements, Options are the possible answers to those requirements, and Criteria refer to the reasoning for choosing a particular option. For instance, we may display the result of a news query of a library customer (the question) either sorted by date of publication, or by author name (the options). The criterion for using either order could be the source of the news item: sort newspaper articles by date, and journal articles by author name.

On one hand, design space analysis results in a rich structure in which an extensive record is built up of the rationale for a specific solution. Why is this system built the way it is? Which alternatives did we consider but reject? Which requirements survived the tradeoffs we had to make? On the other hand, capturing all this information is expensive, and the immediate benefits are difficult to prove, if at all. This is a main reason why design rationale by and large failed to transfer to practice.

9.3 Requirements Specification Techniques

The document that is produced during requirements engineering -- the requirements specification -- serves two groups of people. For the user, the requirements specification is a clear and precise description of the functionality that the system has to offer. For the designer, it is the starting point for the design. It is not easy to serve both groups with one and the same document.

The user is in general best served by a document which speaks his language, the language that is used within the application domain. In the example used before, this would result in using terms like 'title description' and 'catalog'.

The designer on the other hand, is best served with a language in which concepts from his world are used. In terms of the library example, he may prefer concepts like records (an instance of which might be termed 'title description') or files. In one sense, this boils down to a difference in language. However, this difference is of fundamental importance with respect to the later use of the system's description.

If the system is described in the user's language, the requirements specification is mostly phrased in some natural language. If we try to somewhat formalize this description, we may end up with a technique in which certain forms have to be filled in or certain drawing techniques have to be applied.

If, on the other hand, the expert language of the software engineer plays a central role, we often use some formal language. A requirements specification phrased in such a formal language may be checked using formal techniques, for instance with regard to consistency and completeness.

In practice, an outspoken prevalence for the user's expert language shows itself. We may then use existing concepts from the environment in which the system is going to be used. Admittedly, these concepts are not sharply defined, but in general there are no misconceptions between the experts in the application domain as regards the meaning of those concepts. A description in terms of those concepts can thus still be very *precise*. Since the first goal of the requirements specification is to get a *complete* description of the problem to be solved, the user's expert language then would be the best language for the requirements specification.

However, there are certain drawbacks attached to the use of natural language. Meyer (1985) gives an example which illustrates very well what may go wrong when natural language is used in a requirements specification. Meyer lists seven sins which may beset the analyst when using natural language:

- **Noise** This refers to the presence of text elements that do not contain information relevant to the problem. Variants hereof are redundancy and regret. Redundancy occurs when things are repeated. Since natural language is very flexible, related matters can easily be phrased in completely different ways. When this happens the cohesion between matters gets blurred. Regret occurs when statements are reversed or shaded. In the library example, for instance, we could have used the phrase 'a list of all books written by author D' several times and only then realize that this list may be empty, necessitating some special reaction from the system.

- **Silence** Silence occurs when aspects that are of importance for a proper solution of the problem, are not mentioned. An example of this was that the need for two variants on an author's name was not stated explicitly.

- **Over-specification** This occurs when elements of a requirements specification correspond to aspects of a possible solution, rather than to aspects of the problem. As an example, we could have specified that books be kept sorted by the first author's name. Over-specification limits the solution space for the designer.

- **Contradictions** If the description of one and the same aspect is given more than once, in different words, contradictions may occur. This risk is especially threatening when one tries to be too literary. A requirements specification is not meant to be a novel.

- **Ambiguity** Natural language allows for more than one meaning for one and the same phrase. Ambiguity can easily occur when terms are used that belong to the jargon of one or both parties. A 'book' may both denote a physical object and a more abstract entity of which several instantiations (copies) may exist.

- **Forward references** References to aspects of the problem that are only defined later on in the text. This especially occurs in large documents that lack a clear structure. Natural language in itself does not enforce a clear structure.

- **Wishful thinking** A description of aspects of the system such that a realistic solution will be hard to find.

A possible alternative given by Meyer is to first describe and analyze the problem using some formal notation and then translate it back into natural language. The natural language description thus obtained will in general represent a more precise notion of the problem. And it is readable to the user. Obviously, both these models must now be kept up-to-date.

Quite a number of techniques and accompanying notations have evolved to support the requirements engineering process. Most often, the representation generated is a set of semantic networks. Each such representation has various types of nodes and links between nodes, distinguished by visual clues such as their shape or natural language labels. Nodes typically represent things like processes, stores, objects, and attributes. Nodes are joined by arrows representing relationships such as data flow, control flow, abstraction, part-whole, or is-part-of.

Typical examples of such techniques and their representations are discussed in chapter 10. Entity--Relationship Modeling (section 10.1.1) is a widely-known technique to model the data aspect of an information system. Finite State Machines (section 10.1.2) can be viewed as a technique to model the functional aspect. They have a much wider applicability though, and constitute a basic underlying mechanism for many modeling techniques. In particular, the Unified Modeling Language (UML) owes tribute to them. UML diagrams (section 10.3) are widely used to model the result of both requirements engineering and design. Section 9.3.1 touches upon the issue of how to specify non-functional requirements.

9.3.1 Specifying Non-Functional Requirements

The IEEE framework depicted in figure 9.11 lists four types of non-functional requirements: external interface requirements, performance requirements, design constraints and software system attributes. These non-functional requirements can be viewed as constraints placed upon the development process or the products to be delivered.

External interface requirements and design constraints are generally phrased in terms of (non-negotiable) obligations to be met. They are dictated at the start of the project and often concern matters which surpass an individual development project. Examples of such requirements include:

- hardware, software and communications interfaces to be complied with;
- user interfaces that have to obey company standards;
- report formats to be adhered to;
- process constraints such as ISO 9000 compliance or a prescribed development method;

– hardware limitations caused by the available infrastructure.

The remaining non-functional requirements are also known as quality requirements. Quality requirements are notoriously difficult to specify and verify. This topic is dealt with extensively in chapter 6. At this point, we merely wish to re-emphasize two essential issues: quality requirements should be expressed in objective, measurable terms and perfection incurs infinite cost.

Like all other requirements, quality requirements should be verifiable. Requirements such as 'the system should be flexible', 'the system should be user-friendly', or 'response times should be fast', can never be verified and should not therefore appear in the requirements specification. Other phraseology can be used such as 'for activities of type A the system should have a maximum response time of one second in 80% of the cases, while a maximum response time of three seconds is allowed in the remaining 20% of the cases'.

Conversely, extreme levels of quality requirements, such as zero defects or response times of less than 1 second in 100% of the cases, generally incur extremely high costs, or are not feasible at all. Given the fact that users find it difficult to express their true requirements, they may be inclined to ask for too much where quality requirements are concerned, just 'to be on the safe side'. To the analyst and developers, it is likewise difficult to assess the feasibility of those requirements. How can we be sure about response times before even one line of code has been written?

Consider the following example of what may and may not be technically feasible. Suppose we have an application in which two kinds of transactions may occur. Those transactions are characterized by their frequency, CPU-time needed and the number of physical I/O transports. The average I/O access time is also given. Using a statistical distribution describing the dynamics of these systems, one may then answer questions such as: 'how much capacity should the CPU have in order to achieve a response time of at most 2 seconds in X% of cases?' Some given configuration may satisfy the constraints for the case X = 80. A somewhat more stringent requirement (X = 90) may require doubling the CPU-capacity. An even more severe requirement (X = 95) might well not be achievable by the range of machines available.

At first sight, the differences between these requirements seem marginal. They turn out to have a tremendous effect, though. An early and careful analysis of the technical feasibility may yield surprising answers to a number of important questions. Usually, this type of analysis is done at software architecture time. There are many examples of projects in which lots of money was spent on software development efforts which turned out to be not practically feasible (Baber, 1982).

9.4 Verification and Validation

In chapter 1, we argued that a careful study of the correctness of the decisions made at each stage is a critical success factor. This means that during requirements

engineering we should already start verifying and validating the decisions laid down in the requirements specification.

The requirements specification should reflect the mutual understanding of the problem to be solved by the prospective users and the development organization: has everything been described, and has it been described properly. Validating the requirements thus means checking them for properties like correctness, completeness, ambiguity, and internal and external consistency. Of necessity, this involves user participation in the validation process. They are the owners of the problem and they are the only ones to decide whether the requirements specification adequately describes their problem.

If the requirements specification itself is expressed in a formal language, the syntax and semantics of that representation can be verified through formal means. However, the requirements specification can never be completely validated in a formal way, simply because the point of departure of requirements engineering is informal. Most of the testing techniques applied at this stage are therefore informal as well. They are meant to ascertain that the parties involved have the same, proper understanding of the problem. A major stumbling block to this stage is ensuring the user understands the contents of the requirements specification. The techniques applied at this stage often resolve to a translation of the requirements into a form palatable to user inspection: natural-language paraphrasing, the discussion of possible usage scenarios, prototyping, and animation.

Besides testing the requirements specification itself, we also generate at this stage the test plan to be used during system or acceptance testing. A test plan is a document prescribing the scope, approach, resources, and schedule of the testing activities. It identifies the items and features to be tested, the testing tasks to be performed, and the personnel responsible for these tasks. We may at this point develop such a plan for the system testing stage, i.e. the stage at which the development organization tests the complete system against its requirements. Acceptance testing is similar, but is performed under supervision of the user organization. Acceptance testing is meant to determine whether or not the users accept the system.

A more elaborate treatment of the various verification and validation techniques will be given in chapter 13.

9.5 Summary

During requirements engineering we try to get a complete and clear description of the problem to be solved and the constraints that must be satisfied by any solution to that problem. During this phase, we do not only consider the functions to be delivered, but we also pay attention to requirements imposed by the environment. The requirements engineering phase results in a series of models concentrating on different aspects of the system (such as its functionality, user interface and communication structure) and different perspectives (audiences). The result of this process is documented in a requirements specification. A good framework for the contents of the requirements

specification is given in (IEEE830, 1993). It should be kept in mind that this document contains an a posteriori reconstruction of an as yet ill-understood iterative process.

This iterative process involves four types of activity:

- requirements elicitation, which is about *understanding* the problem,
- requirements specification, which is about *describing* the problem,
- requirements validation, which is about *agreeing upon* the problem, and
- requirements negotiation, which is about *fitting* the problem to the situation at hand.

In many cases, fully completing requirements engineering before design and construction start is not feasible. In agile development processes, requirements engineering is intertwined with design and construction.

During requirements engineering we are modeling part of reality. The part of reality we are interested in is referred to as the universe of discourse (UoD). The modeling process is termed conceptual modeling.

People involved in a UoD have an implicit conceptual model of that UoD. During conceptual modeling, an implicit model is turned into an explicit one. The explicit conceptual model is used to communicate with other people, such as users and designers, and to assess the validity of the system under development during all subsequent phases. During the modeling process, the analyst is confronted with two types of problem: analysis problems and negotiation problems. Analysis problems have to do with getting the requirements right. Negotiation problems arise because different people involved may have different views on the UoD to be modeled, opposing interests, and so on.

Existing approaches to requirements engineering are largely Taylorian in nature. They fit a functional view of software development in which the requirements engineering phase serves to elicit the 'real' user requirements. It is increasingly being recognized that the Taylorian approach need not be the most appropriate approach to requirements engineering. Many UoDs under consideration involve people whose world model is incomplete, irrational, or in conflict with the world view of others. In such cases, the analyst is not a passive outside observer of the UoD, but actively participates in shaping the UoD. The analyst gets involved in negotiation problems and has to choose the view of some party involved, or assist in obtaining some compromise.

The following description techniques are often used for the requirements specification:

- natural language,
- pictures, and
- formal language.

An advantage of using natural language is that the specification is very readable and understandable to the user and other non-professionals involved. Pictures may be put to advantage in bringing across the functional architecture of the system. A formal language allows us to use tools in analyzing the requirements. Because of its precision, it is a good starting point for the design phase. We may also argue that both formal and informal notations be used, since they augment and complement each other. For each of the parties involved, a notation should be chosen that is appropriate to the task at hand.

9.6 Further Reading

There are many text books fully devoted to requirements engineering. Davis (1993) provides a fairly complete coverage of 'classic' requirements specification techniques. Davis (2005) focuses on requirements engineering in the face of tight schedule constraints. Wieringa (1996) discusses a number of requirements specification techniques in quite some depth. The distinction between implicit and explicit conceptual models is made there too. Loucopoulos and Karakostas (1995) and Kotonya and Sommerville (1997) have a stronger emphasis on the full requirements engineering process. Juristo et al. (2002) discuss the state of the practice in requirements engineering. Hofman and Lehner (2001) focus on successful requirements practices. Sommerville (2005) discusses recent developments in the field.

Pohl (1993) and Goguen and Jirotka (1994) emphasize the role of social and cognitive issues in requirements engineering. Ramos et al. (2005) argue that emotion is relevant in requirements engineering. The thin spread of application knowledge amongst the specialists involved is discussed in (Curtis et al., 1988). Difficulties of requirements engineering for market-driven software development are addressed in (Potts, 1993). Moynihan (2000) discusses how managers cope with requirements uncertainty.

The objectivist--subjectivist and order--conflict dimensions and the resulting four paradigms for requirements engineering are discussed in (Hirschheim and Klein, 1989). Various socio-technical, subjectivist, approaches to requirements elicitation are discussed in (Atkinson, 2000).

Task analysis is discussed in (Sebillotte, 1988). Scenario-based requirements engineering techniques are discussed in (Weidenhaupt et al., 1998) and (TrSE, 1998). Sutcliffe et al. (1998) describe how to create and document scenarios during requirements engineering. Business Process Redesign is described in (Keen, 1991) and (Tapscott and Caston, 1993). A framework for BPR is given in (Davenport, 1993).

Research in requirements elicitation is aimed at developing techniques which overcome our limitations as humans in conveying information. An early overview of this type of problem is given in (Davis, 1982). A more recent survey and evaluation of elicitation techniques is given in (Goguen and Linde, 1993). Example experience reports are given in (Sommerville et al., 1994) and (Coakes and Coakes,

2000) (ethnographic approach) and (Beyer and Holtzblatt, 1995) (the analyst as an apprentice to the user).

Lamsweerde (2001) gives a very good overview of goal-oriented requirements engineering. Mylopoulos et al. (2001) is another article by pioneers in this area. Darke and Shanks (1996) and Sommerville and Sawyer (1997) provide a good overview of viewpoints in the context of requirements engineering.

Leffingwell and Widrig (2000) is fully devoted to requirements management. One of the first discussions of requirements traceability is (Gotel and Finkelstein, 1994). Design space analysis is discussed in (Moran and Carroll, 1994). Questions, Options, Criteria (QOC) stem from (MacLean et al., 1991). gIBIS, a hypertext system designed to capture early design decisions, is described in (Conklin and Begeman, 1988).

Kano's model is discussed in (Kano, 1993). The quest for creativity in requirements engineering is further stressed in (Robertson, 2002), Austin and Devin (2003) and (Maiden et al., 2004).

Morisio et al. (2002) gives a taxonomy of COTS products. COTS selection procedures are discussed in (Maiden and Ncube, 1998).

Exercises

1. What are the four major types of activity in requirements engineering?
2. What is requirements elicitation?
3. What is the difference between an implicit and an explicit conceptual model?
4. In what sense are most requirements engineering techniques Taylorian in nature?
5. Describe the requirements elicitation technique called task analysis.
6. Describe the requirements elicitation technique called scenario-based analysis.
7. In which circumstances is ethnography a viable requirements elicitation technique?
8. What is goal-oriented requirements engineering?
9. How can conflicting requirements be represented in viewpoints?
10. What does MoSCoW stand for?
11. Why is the distinction between 'Attractive', 'Must-be' and 'One-dimensional' categories of requirements in Kano's model relevant?

12. How does the presence of COTS components affect requirements engineering?

13. Why is requirements traceability important?

14. List and discuss the major quality requirements for a requirements document.

15. List and discuss major drawbacks of using natural language for specifying requirements.

16. ♠ Draw up a requirements specification for a system whose development you have been involved with, following IEEE 830. Discuss the major differences between the original specification and the one you wrote.

17. ♡ What are major differences in the external environment of an office automation system and that of an embedded system, like an elevator control system. What impact will these differences have on the requirements elicitation techniques to be employed?

18. ♡ For an office information system, identify different types of stakeholders. Can you think of ways in which the requirements of these stakeholders might conflict?

19. ♡ Refine the framework in figure 9.1 such that it reflects the situation in which we have to explicitly model both the current and the new work situation.

20. ♡ Discuss pros and cons of the following descriptive means for a requirements specification: full natural language, constrained natural language, a pictorial language like UML.

21. ♡ Which of the descriptive means mentioned in the previous exercise would you favor for describing the requirements of an office automation system? And which one for an elevator control system?

22. ♠ Take the requirements specification document from a project you have been involved in and assess it with respect to the requirements for such a document as listed in section 9.2 (unambiguity, completeness, etc.).

23. ♡ How would you test the requirements stated in the document from the previous exercise? Are the requirements testable to start with?

24. ♠ How would you go about determining the requirements for a hypertext-like browsing system for a technical library. Both users and staff of the library only have experience with keyword-based retrieval systems.

25. ♡ As an analyst involved in the development of this hypertext browsing

system, discuss possible stands in the subjectivist--objectivist and order--conflict dimensions. What are the arguments for and against these stands?

26. ♠ Write a requirements specification for a hypertext browsing system.

27. ♡ Study the following specification for a simple line formatter:

The program's input is a stream of characters whose end is signaled with a special end-of-text character, ET. There is exactly one ET character in each input stream. Characters are classified as:

- break characters -- BL (blank) and NL (new line);
- nonbreak characters -- all others except ET;
- the end-of-text indicator - ET.

A *word* is a non-empty sequence of nonbreak characters. A *break* is a sequence of one or more break characters. Thus, the input can be viewed as a sequence of words separated by breaks, with possible leading and trailing breaks, and ending with ET.

The program's output should be the same sequence of words as in the input, with the exception that an oversize word (i.e. a word containing more than MAXPOS characters, where MAXPOS is a positive integer) should cause an error exit from the program (i.e. a variable, Alarm, should have the value TRUE). Up to the point of an error, the program's output should have the following properties:

1 A new line should start only between words and at the beginning of the output text, if any.

2 A break in the input is reduced to a single break character in the output.

3 As many words as possible should be placed on each line (i.e. between successive NL characters).

4 No line may contain more than MAXPOS characters (words and BLs).

Identify as many trouble spots as you can in this specification. Compare your findings with those in (Meyer, 1985).

28. ♡ What are the major uses of a requirements specification. In what ways do these different uses affect the style and contents of a requirements document?