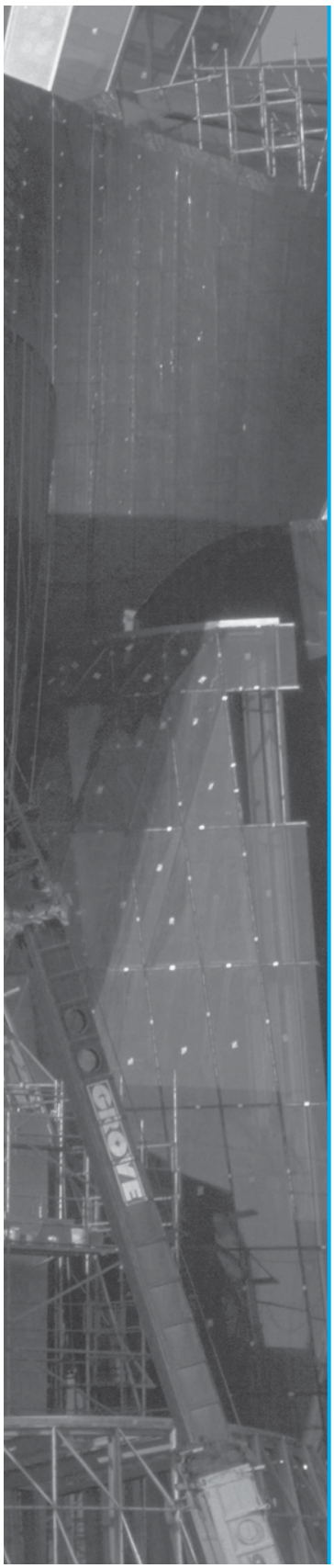



# 4

# Requirements engineering

## Objectives

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

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

## Contents

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

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

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

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

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

Different levels of requirements are useful because they communicate information about the system to different types of reader. Figure 4.1 illustrates the distinction between user and system requirements. This example from a mental health care patient management system (MHC-PMS) shows how a user requirement may be expanded into several system requirements. You can see from Figure 4.1 that the user requirement is quite general. The system requirements provide more specific information about the services and functions of the system that is to be implemented.

**User Requirement Definition**

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

**System Requirements Specification**

**1.1** On the last working day of each month, a summary of the drugs prescribed, their cost, and the prescribing clinics shall be generated.
**1.2** The system shall automatically generate the report for printing after 17.30 on the last working day of the month.
**1.3** A report shall be created for each clinic and shall list the individual drug names, the total number of prescriptions, the number of doses prescribed, and the total cost of the prescribed drugs.
**1.4** If drugs are available in different dose units (e.g., 10 mg, 20 mg) separate reports shall be created for each dose unit.
**1.5** Access to all cost reports shall be restricted to authorized users listed on a management access control list.

**Figure 4.1** User and system requirements

You need to write requirements at different levels of detail because different readers use them in different ways. Figure 4.2 shows possible readers of the user and system requirements. The readers of the user requirements are not usually concerned with how the system will be implemented and may be managers who are not interested in the detailed facilities of the system. The readers of the system requirements need to know more precisely what the system will do because they are concerned with how it will support the business processes or because they are involved in the system implementation.

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

## 4.1 Functional and non-functional requirements

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

1. *Functional requirements* These are statements of services the system should provide, how the system should react to particular inputs, and how the system

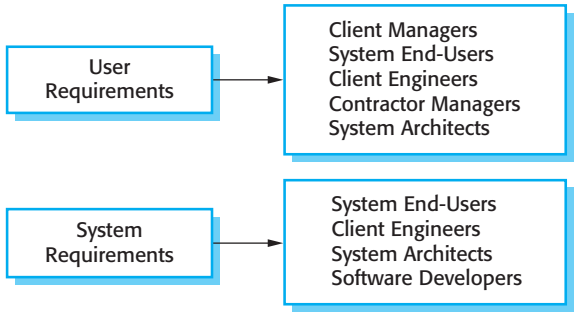



**Figure 4.2** Readers of different types of requirements specification

should behave in particular situations. In some cases, the functional requirements may also explicitly state what the system should not do.

2. *Non-functional requirements* These are constraints on the services or functions offered by the system. They include timing constraints, constraints on the development process, and constraints imposed by standards. Non-functional requirements often apply to the system as a whole, rather than individual system features or services.

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

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

### 4.1.1 Functional requirements

The functional requirements for a system describe what the system should do. These requirements depend on the type of software being developed, the expected users of the software, and the general approach taken by the organization when writing requirements. When expressed as user requirements, functional requirements are usually described in an abstract way that can be understood by system users. However, more specific functional system requirements describe the system functions, its inputs and outputs, exceptions, etc., in detail.

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

**Domain requirements**

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

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

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

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

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

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

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

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

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

In principle, the functional requirements specification of a system should be both complete and consistent. Completeness means that all services required by the user should be defined. Consistency means that requirements should not have contradictory

definitions. In practice, for large, complex systems, it is practically impossible to achieve requirements consistency and completeness. One reason for this is that it is easy to make mistakes and omissions when writing specifications for complex systems. Another reason is that there are many stakeholders in a large system. A stakeholder is a person or role that is affected by the system in some way. Stakeholders have different—and often inconsistent—needs. These inconsistencies may not be obvious when the requirements are first specified, so inconsistent requirements are included in the specification. The problems may only emerge after deeper analysis or after the system has been delivered to the customer.

### 4.1.2 Non-functional requirements

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

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

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

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

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

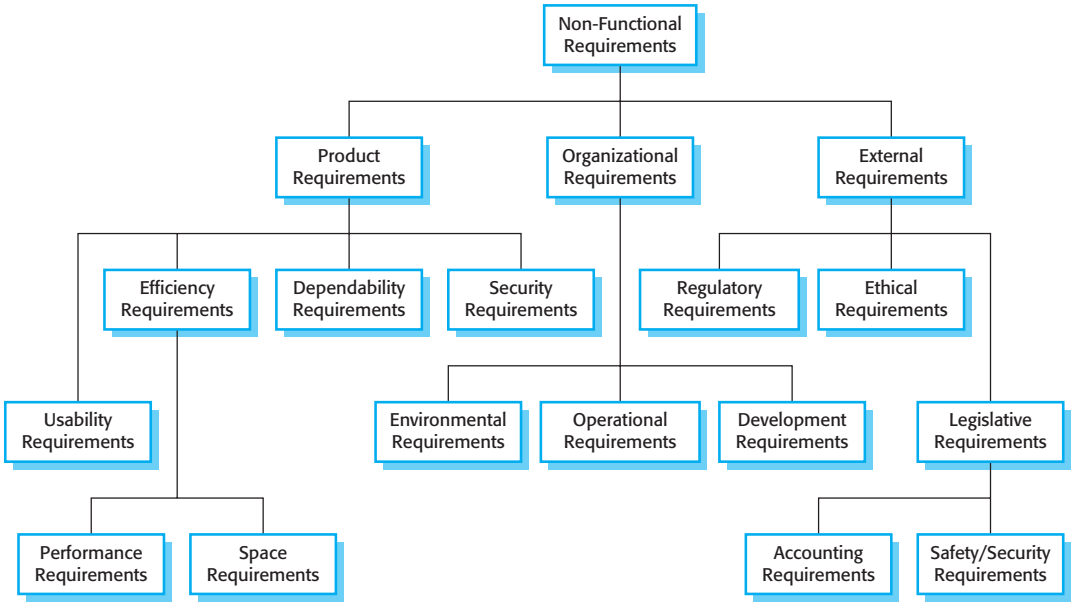



**Figure 4.3** Types of non-functional requirement

hardware systems, or external factors such as safety regulations or privacy legislation. Figure 4.3 is a classification of non-functional requirements. You can see from this diagram that the non-functional requirements may come from required characteristics of the software (product requirements), the organization developing the software (organizational requirements), or from external sources:

1. *Product requirements* These requirements specify or constrain the behavior of the software. Examples include performance requirements on how fast the system must execute and how much memory it requires, reliability requirements that set out the acceptable failure rate, security requirements, and usability requirements.
2. *Organizational requirements* These requirements are broad system requirements derived from policies and procedures in the customer's and developer's organization. Examples include operational process requirements that define how the system will be used, development process requirements that specify the programming language, the development environment or process standards to be used, and environmental requirements that specify the operating environment of the system.
3. *External requirements* This broad heading covers all requirements that are derived from factors external to the system and its development process. These may include regulatory requirements that set out what must be done for the system to be approved for use by a regulator, such as a central bank; legislative requirements that must be followed to ensure that the system operates within the law; and ethical requirements that ensure that the system will be acceptable to its users and the general public.

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

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

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

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

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

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

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

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

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

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

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

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

**Figure 4.5** Metrics for specifying non-functional requirements

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

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

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

It is difficult, in practice, to separate functional and non-functional requirements in the requirements document. If the non-functional requirements are stated separately from the functional requirements, the relationships between them may be hard to understand. However, you should explicitly highlight requirements that are clearly related to emergent system properties, such as performance or reliability. You can do this by putting them in a separate section of the requirements document or by distinguishing them, in some way, from other system requirements.

**Requirements document standards**

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

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

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

## 4.2 The software requirements document

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

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

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

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

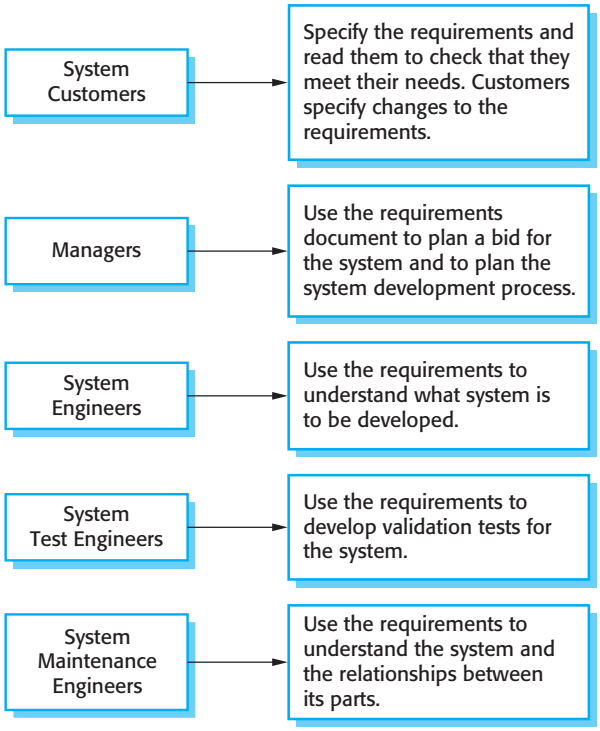



**Figure 4.6** Users of a requirements document

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

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

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

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

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

**Figure 4.7** The structure of a requirements document

requirements document will leave out many of detailed chapters suggested above. The focus will be on defining the user requirements and high-level, non-functional system requirements. In this case, the designers and programmers use their judgment to decide how to meet the outline user requirements for the system.

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

**Problems with using natural language for requirements specification**

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

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

to a fine level of detail. This means that the requirements documents are likely to be very long and should include most if not all of the chapters shown in Figure 4.7. For long documents, it is particularly important to include a comprehensive table of contents and document index so that readers can find the information that they need.

## 4.3 Requirements specification

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

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

System requirements are expanded versions of the user requirements that are used by software engineers as the starting point for the system design. They add detail and explain how the user requirements should be provided by the system. They may be used as part of the contract for the implementation of the system and should therefore be a complete and detailed specification of the whole system.

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

1. You may have to design an initial architecture of the system to help structure the requirements specification. The system requirements are organized according to

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

**Figure 4.8** Ways of writing a system requirements specification

the different sub-systems that make up the system. As I discuss in Chapters 6 and 18, this architectural definition is essential if you want to reuse software components when implementing the system.

2. In most cases, systems must interoperate with existing systems, which constrain the design and impose requirements on the new system.
3. The use of a specific architecture to satisfy non-functional requirements (such as N-version programming to achieve reliability, discussed in Chapter 13) may be necessary. An external regulator who needs to certify that the system is safe may specify that an already certified architectural design be used.

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

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

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

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

**Figure 4.9** Example requirements for the insulin pump software system

### 4.3.1 Natural language specification

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

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

1. Invent a standard format and ensure that all requirement definitions adhere to that format. Standardizing the format makes omissions less likely and requirements easier to check. The format I use expresses the requirement in a single sentence. I associate a statement of rationale with each user requirement to explain why the requirement has been proposed. The rationale may also include information on who proposed the requirement (the requirement source) so that you know whom to consult if the requirement has to be changed.
2. Use language consistently to distinguish between mandatory and desirable requirements. Mandatory requirements are requirements that the system must support and are usually written using ‘shall’. Desirable requirements are not essential and are written using ‘should’.
3. Use text highlighting (bold, italic, or color) to pick out key parts of the requirement.
4. Do not assume that readers understand technical software engineering language. It is easy for words like ‘architecture’ and ‘module’ to be misunderstood. You should, therefore, avoid the use of jargon, abbreviations, and acronyms.
5. Whenever possible, you should try to associate a rationale with each user requirement. The rationale should explain why the requirement has been included. It is particularly useful when requirements are changed as it may help decide what changes would be undesirable.

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

***Insulin Pump/Control Software/SRS/3.3.2***

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

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

### 4.3.2 Structured specifications

Structured natural language is a way of writing system requirements where the freedom of the requirements writer is limited and all requirements are written in a standard way. This approach maintains most of the expressiveness and understandability of natural language but ensures that some uniformity is imposed on the specification. Structured language notations use templates to specify system requirements. The specification may use programming language constructs to show alternatives and iteration, and may highlight key elements using shading or different fonts.

The Robertsons (Robertson and Robertson, 1999), in their book on the VOLERE requirements engineering method, recommend that user requirements be initially written on cards, one requirement per card. They suggest a number of fields on each card, such as the requirements rationale, the dependencies on other requirements, the source of the requirements, supporting materials, and so on. This is similar to the approach used in the example of a structured specification shown in Figure 4.10.

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

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

**Figure 4.11** Tabular specification of computation for an insulin pump

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

1. A description of the function or entity being specified.
2. A description of its inputs and where these come from.
3. A description of its outputs and where these go to.
4. Information about the information that is needed for the computation or other entities in the system that are used (the 'requires' part).
5. A description of the action to be taken.
6. If a functional approach is used, a pre-condition setting out what must be true before the function is called, and a post-condition specifying what is true after the function is called.
7. A description of the side effects (if any) of the operation.

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

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

Tables are particularly useful when there are a number of possible alternative situations and you need to describe the actions to be taken for each of these. The insulin pump bases its computations of the insulin requirement on the rate of change of blood sugar levels. The rates of change are computed using the current and previous readings. Figure 4.11 is a tabular description of how the rate of change of blood sugar is used to calculate the amount of insulin to be delivered.

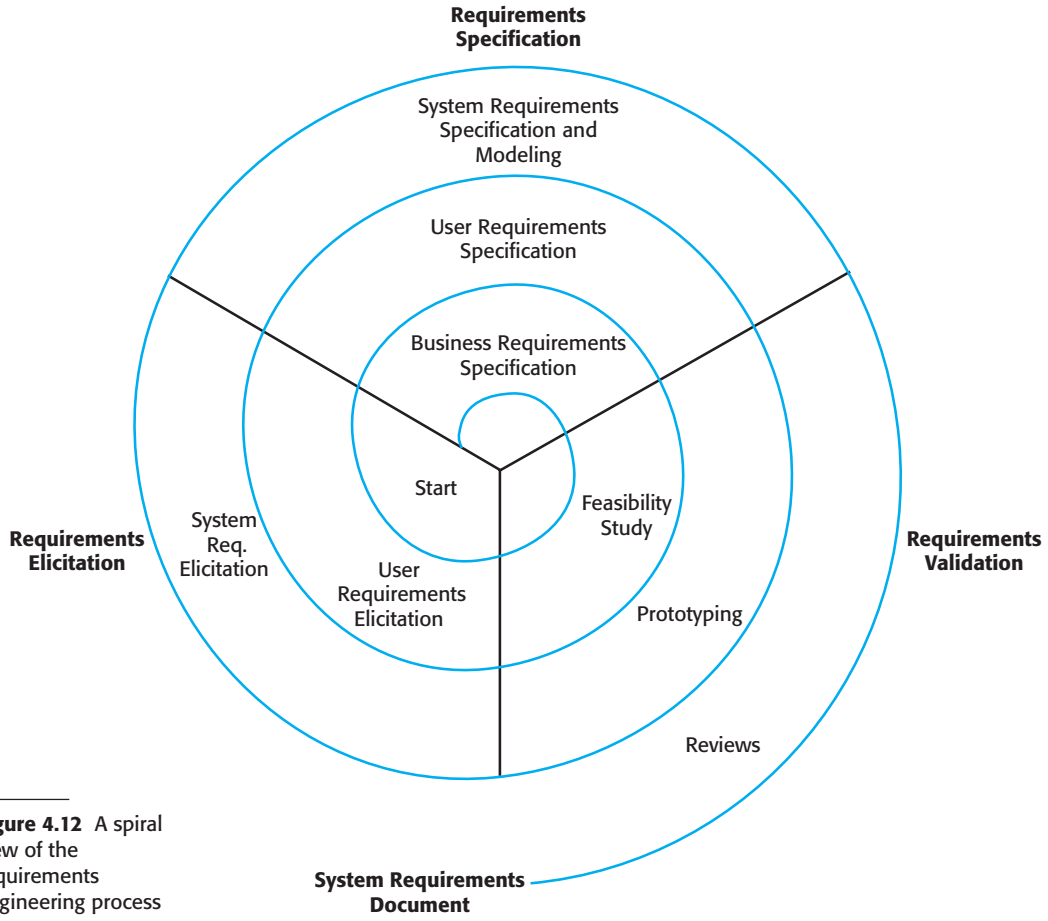



**Figure 4.12** A spiral view of the requirements engineering process

## 4.4 Requirements engineering processes

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

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

**Feasibility studies**

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

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

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

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

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

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

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

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

## 4.5 Requirements elicitation and analysis

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

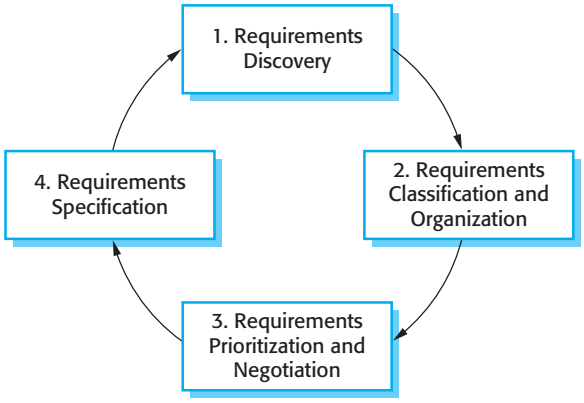



**Figure 4.13** The requirements elicitation and analysis process

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

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

The process activities are:

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

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

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

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

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

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

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

**Viewpoints**

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

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

### 4.5.1 Requirements discovery

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

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

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

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

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

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

### 4.5.2 Interviewing

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

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

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

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

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

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

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

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

Effective interviewers have two characteristics:

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

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

### 4.5.3 Scenarios

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

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

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

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

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

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

The nurse chooses the menu option to add medical history.

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

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

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

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

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

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

**Figure 4.14** Scenario for collecting medical history in MHC-PMS

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

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

### 4.5.4 Use cases

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

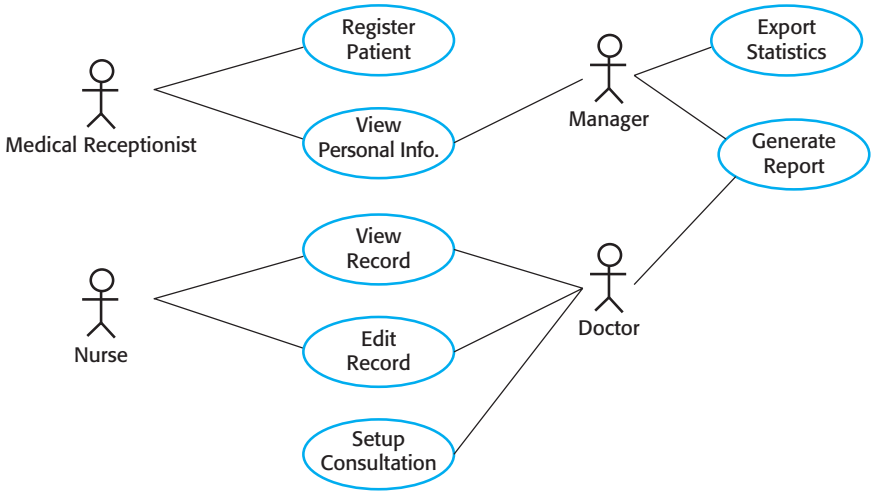



**Figure 4.15** Use cases for the MHC-PMS

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

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

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

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

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

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

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

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

### 4.5.5 Ethnography

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

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

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

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

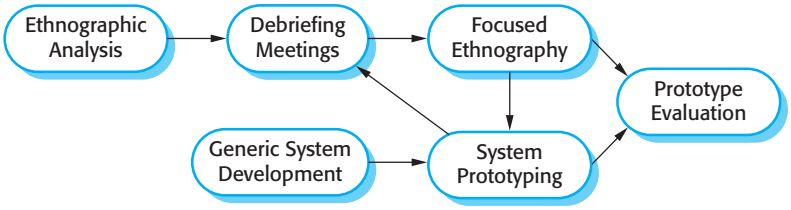



**Figure 4.16** Ethnography and prototyping for requirements analysis

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

Ethnography is particularly effective for discovering two types of requirements:

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

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

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

**Requirements reviews**

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

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

## 4.6 Requirements validation

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

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

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

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

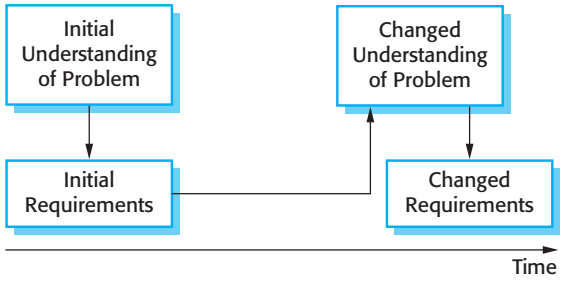



**Figure 4.17** Requirements evolution

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

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

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

## 4.7 Requirements management

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

**Enduring and volatile requirements**

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

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

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

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

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

### 4.7.1 Requirements management planning

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

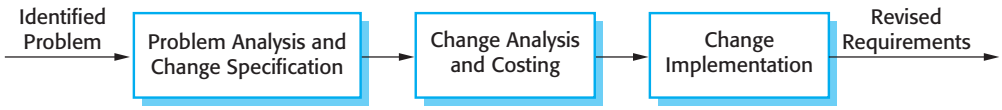



**Figure 4.18** Requirements change management

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

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

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

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

### 4.7.2 Requirements change management

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

**Requirements traceability**

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

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

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

There are three principal stages to a change management process:

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

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

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

## KEY POINTS

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

## FURTHER READING

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

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

*Mastering the Requirements Process, 2nd edition*. A well-written, easy-to-read book that is based on a particular method (VOLERE) but which also includes lots of good general advice about requirements engineering. (S. Robertson and J. Robertson, 2006, Addison-Wesley.)

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

## EXERCISES

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

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

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

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

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

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

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

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

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

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

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

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

## REFERENCES

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Softw Syst Model (2003) 2: 5–14 / Digital Object Identifier (DOI) 10.1007/s10270-003-0020-3

# Models in software engineering – an introduction

**Jochen Ludewig**

Institute of Software Technology at Stuttgart University, Breitwiesenstr. 20–22, 70565 Stuttgart, Germany;
E-mail: ludewig@informatik.uni-stuttgart.de

Received: 21 October 2002/Accepted: 10 January 2003
Published online: 27 February 2003 – 

**Abstract.** Modelling is a concept fundamental for software engineering. In this paper, the word is defined and discussed from various perspectives. The most important types of models are presented, and examples are given.

Models are very useful, but sometimes also dangerous, in particular to those who use them unconsciously. Such problems are shown. Finally, the role of models in software engineering research is discussed.

**Keywords:** Models – Software engineering – Metaphors – SESAM

## 1 Disclaimer and goals

We use models when we think about problems, and when we talk to each other, and when we construct mechanisms, and when we try to understand phenomena, and when we teach. In short, we use models all the time.

That means: models have never been invented, they have been around (at least) since humans started to exist. Therefore, nobody can just define what a model is, and expect that other people will accept this definition; endless discussions have proven that there is no consistent common understanding of models.

In this paper, this difficulty is (almost) ignored, and the term "model" is defined as if there were no conflicting opinions. This approach is taken because it is the only way (at least the only way known to the humble author) for investigating the power and the limitation of models. Readers are not required to accept my definitions permanently; but they might be prepared to accept them at least while they read this contribution, because my judgements and conclusions are based on my definitions.

This paper is intended to help the reader

- recognize models where they appear,
- know the properties and the power of models,
- clearly distinguish models from the original objects,
- create new models where appropriate.

## 2 The air we breathe

Our ability for modelling is not acquired but given to us from birth. Without it, we would not be able to reduce the vast flow of information to a rate we can cope with. By mapping visible and invisible phenomena to *notions* (in German: *Begriffe*), the number of *different* observations is significantly reduced, and we deal with some classes of problems rather than with millions of individual problems. Hence, we can collect experiences, find generic solutions and decisions, and develop strategies for surviving in the real world. The ability for reflection, which is considered *the* difference between man and animal, is directly related to the use of models.

The particular strength of models is based on the idea of *abstraction*: a model is usually not related to one particular object or phenomenon only, but to many, possibly to an unlimited number of them, it is related to a *class*. They who note that the change from high tide to low tide and from low tide to high tide follows a certain rhythm can prepare for, or make use of it. Those who learn that a certain class of animals rather than one single living creature is fast, strong, and dangerous, have improved their chances for survival.

While we live, i.e. act and react, we use models all the time, usually unconsciously. The situation is quite different in research and engineering: there, the creation of models is an explicit topic; it is the purpose of research and an important step in producing artefacts. Research yields theories. A theory is a special kind of model (see below). The more influential a theory is in the world, the higher it is estimated. Such influence ranges from a change of our perception of the world (like switching

from a geocentric to a heliocentric view) to a massive effect and threat like that of nuclear bombs.

The role of modelling in engineering is similar. Models help in developing artefacts by providing information about the consequences of building those artefacts before they are actually made. Such information may be highly formal (like the theory of mechanics, as it is applied in the construction of bridges, or the theory of computational complexity which is used in the analysis of algorithms) or rather informal (like a rough drawing of a machine, or a textual specification of a software system). Interface definitions are models too; they are particularly important for the spreading of technology.

## 3 Definitions

The term "model" is derived from the Latin word *modulus*, which means *measure, rule, pattern, example to be followed*. Obvious examples are toy railways and dolls, maps, architectural models of buildings. In software engineering, we have process models, design patterns, class diagrams. Other models are less obvious, like project plans, specifications and designs, metrics, and minutes of project meetings.

### *3.1 The model criteria*

In order to distinguish models from other artefacts, we need *criteria*. According to Stachowiak [8], any candidate must meet three criteria, or otherwise it is not a model:

- **Mapping criterion**: there is an original object or phenomenon that is mapped to the model. In the sequel, this original object or phenomenon is referred to as "the original".
- **Reduction criterion**: not all the properties of the original are mapped on to the model, but the model is somehow reduced. On the other hand, the model must mirror at least *some* properties of the original.
- **Pragmatic criterion**: the model can replace the original for some purpose, i.e. the model is useful.

The mapping criterion does not imply the actual existence of the original; it may be planned, suspected, or fictitious. The dwarfs and fairies found in many gardens model fictitious creatures, and a map of Troy mirrors a historic theory that is not at all generally accepted. Most novels and movies model a reality that was born in the author's fantasy. The cost estimation of a software project is a speculative model of the future.

A model may act as the original of another model, we can find cascades of models, e.g. when a painting shows a room with paintings. A program design is a model of the code to be written, while the code is a model of the computation performed by the computer when the code is executed.

At first glance, the reduction criterion seems to describe a weakness of models, because something is lost in the model that was present in the original. But that loss is the real strength of models: very often, the model can be handled while the original cannot. A map is handy and cheap. Even when we can use a space-ship (which is neither handy nor cheap), we cannot identify most of the details that are clearly marked in the map.

The pragmatic criterion is the reason why we use models. Since we are not able or not willing to use the original, we use the model instead. That applies to a toy that represents an extremely friendly animal, and it also applies to the theory of a big bang as the birthday of our universe, because no scientist can directly observe what happened billions of years ago.

Figure 1 shows the relationship between original and model. Note that a model is not necessarily similar to the original in any naive sense, like a toy automobile that looks similar to a real automobile. The attributes may be mapped in many different ways. In photography, colours are translated into values on a grey scale. Physical properties may be recorded as numbers. Artists express feelings by shapes and sounds; the daytime is indicated by the angles of two clock hands.

As an effect of the reduction, many features of the original (the *waived attributes*) are not found in the model. For example, the name of a person is not visible in his photograph. On the other hand, features that do not stem from the original are added (*abundant attributes*). For example, the size of the picture does not tell anything about the person. A Z specification does not comprise most of the details later found in the program, but many syntactical details (like arrows etc.) that are given as part of the specification language; their shapes have no meaning for the system to be developed.

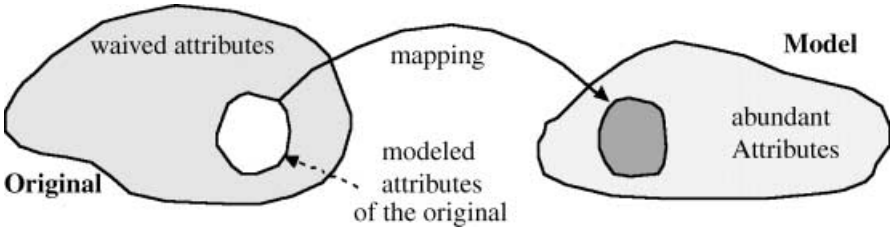



**Fig. 1.** Original and model according to Stachowiak

Let us use an identity card as an example:

The IC is issued for a particular (say female) person (mapping criterion). If the card is faked, that person does not really exist. Most of her properties are waived, like her memories, her taste, or her favourite destination for holidays (reduction criterion). A number gives her height; her residence is indicated by her address. The IC is useful (pragmatic criterion) when somebody wants to find out certain properties of the person who might not be able to answer questions (e.g. after an accident) or not be willing to answer them correctly (e.g. when the person tries to open a banking account).

### *3.2 Related terms*

Many frequently used terms have a meaning that is similar to the meaning of the term "model" (see Fig. 2). Let us discuss which of those terms can be regarded as proper models. There is, of course, no sharp distinction.

**Tool**: many tools, perhaps all tools have been invented as imitations of existing aids. A hammer is an improved copy of the human fist. But a tool is not a model: The relationship to its original very soon becomes obsolete, because the tool is modified again and again, independently from its original. In the early days of computing machines (i.e. two hundred years ago), those machines copied the notes people write on paper when they calculate. Modern computer hardware no longer behaves in any way similar to what people do; a computer is *not* an artificial brain.

**Name**: a name is not a model; while it can be used in place of the object it identifies, it does not provide any information about the object.

**Icons** represent programs and files. Any action on an icon is a substitute for an action on the program or file. But little, if any, information about the original object is contained in the icon. Therefore, icons are not regarded as models. The same is true for **symbols**.

**Metaphor**: describing a complicated machine, we often identify a central part, and refer to it as the *heart* of the machine. When a term from one area (e.g. from biology) is used in a different area (like mechanical machines) in order to describe some characteristics (it keeps the whole thing running) it is called a metaphor.

What about the model criteria? Let us look at the term "software virus" as an example. First, there is an original (the harmful code; note that the biological virus is *not* the original; it only provides the *word* and its *connotation*). Second, the metaphor is an extremely reduced representation of the original. (When somebody tells us about a new virus, the metaphor conveys some information about the type of program, but not about its size or its algorithm). Third, the original is represented by the metaphor, and we can use the metaphor when we think about defence against it. (A virus is dangerous, and may be passed to another victim unintentionally. We can prevent an *epidemic* spread of a virus by keeping the *infected* computer in *isolation*.) A metaphor *is* a special type of model.

Metaphors are particularly useful when we deal with something that is quite new, and not yet named. Computers, including all their details and attributes, were invented only recently, and there was a complete lack of genuine names. Few, if any, new words were introduced; the vast majority of missing names were replaced by metaphors. Therefore, we use the terms *jump*, *loop*, *crash*, *freeze*, *bug*, *overflow, storage* and *memory*, *maintenance*, *firewall*, *protocol*, *message*, *window* etc. etc. Most of these metaphors were presumably created spontaneously, only a few were (probably) invented (like file, desktop, transaction, handshake).

Some metaphors are misleading. A typical example is *inheritance*, as used in object-oriented programming. In its traditional meaning, inheritance applies to individuals: when the parents die, their property is passed on to their children. In biology, the word was used in a similar meaning: parents pass their genetic material to their children (but the material is copied). The meaning of inheritance in object-oriented programming is very different: it applies to classes of objects rather than to objects. And it establishes a permanent dependency: when a class is modified, all dependent classes are implicitly modified too. This may be the reason why so many programmers fail to understand inheritance.

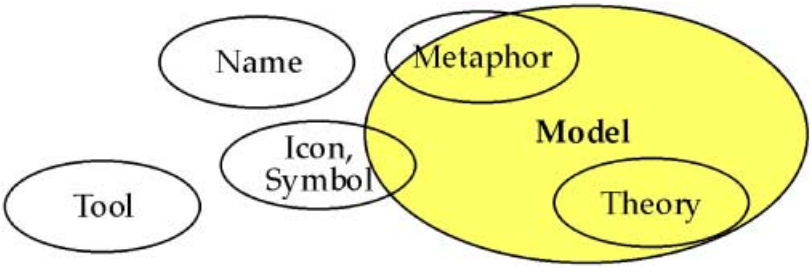



**Fig. 2.** "Model" and related terms

Another example is *maintenance*. While the word implies a conservative approach (maintain the original state), software maintenance is done for the sole purpose of changing the state. The meaning has been inverted!

**Theory**: a theory widely used in computer science is the theory of finite state machines. The finite state machine is a model of the computers we use, or of some parts of our computers. The theory of finite state machines is a model of the behaviour of our computers. The fact that switching takes some time in any physical device (i.e. there are states between the states) is ignored. By applying this theory, we are able to predict the behaviour, and we can construct useful machines. In general, every theory is a model of some phenomena that can be observed and/or induced. A theory is a highly abstract type of model; it emphasizes results and conclusions rather than obviousness.

Note that theories need not to be formal. Brook's law ("Adding manpower to a late project makes it even later") is an example of an informal theory.

## 4 Model taxonomy

### *4.1 Descriptive and prescriptive models*

Models in a narrower sense (i.e. not including metaphors) can be classified under various aspects. A model can mirror an existing original (like a photograph), or it can be used as a specification of something to be created (like a construction plan). In the former case, we call it a *descriptive* model; in the latter case, we call it *prescriptive*. When an architect sketches an old house, and then adds to his drawing some modifications he suggests to the owner, the model is first descriptive, later on prescriptive. We call it a *transient* model.

Though we talk about descriptive or prescriptive models, this is in fact a property of the *relationship* between a (particular) model and a (particular) original rather than a property of the model. A construction plan is not necessarily prescriptive; it may have been produced from an existing object. Therefore, a model may be descriptive with respect to one original, and, at the same time, be prescriptive with respect to another original. A simple example is a drawing of an antique golden ring that is used for reproducing similar rings.

At first glance, it seems obvious that descriptive models can only be created after the original, while prescriptive models need to exist before the original is made. The latter is in fact true, but the former is not (or not precisely) true: prognostic models that describe something that does not yet exist are descriptive, because they are not intended to influence the original. The weather forecast is such a model: though we cannot (yet) influence the weather consciously, we can fairly reliably describe the weather we will see tomorrow. The same is true for a cost estimation, which has no (direct) influence on the actual cost. Such estimation should be clearly distinguished from a requirement ("The cost of the project must not exceed 500 k€"), which is, of course, a prescriptive model. If there is more than one prescriptive model, those models may be inconsistent. There may be a requirements specification that turns out to be inconsistent with the cost limit; then, there is no solution that fits both models.

Descriptive models are applied in order to make some specific information about the original easily and quickly accessible: the table of contents in a documentation reduces the time for finding a specific topic, and the program size (in LOC, e.g.) can be used for simple conclusions, like "if program A is ten times larger than program B, it will probably not fit into the available memory."

### *4.2 Purposes of models*

The purpose of the models is another criterion that can be used for classifying models.

- *Documentation* is created when data in its most general sense is derived from data that is already existing and available. Therefore, it is descriptive. Three important subclasses are
  - *concise descriptions* as stored in software databases or included in tenders and marketing information
  - *minutes, protocols*, like the log of a test, or of the dialogues in which the customer describes his or her requirements. The requirements specification (which is prescriptive) mirrors those requirements.
  - *metrics*, both *software metrics* and *process metrics*, are highly abstract models. The number of pages of a specification, the number of classes in an object-oriented program, the response time of a program, and the memory it occupies are examples of software (or product) metrics. The duration and cost of development and the number of developers are two important process metrics.
- *Instructions* (like "copy the file to your harddisc") provide information about some activity (like installation, or test). Instructions are prescriptive.
- *Explorative models* are transient (see Fig. 3). They are used when the consequences of a change are to be evaluated. The modifications are applied to the model rather than to the real system. When their effect seems to be positive, the modifications are applied to the original. If we are not sure about a new interface of an information system, we implement a prototype that can be evaluated, even though it does not provide the functionality of the target system.
- *Educational models* and *games* replace the originals for ethical or practical reasons. Examples are models of the human body as used in medical education, flight simulators, and the dolls children play with. All these

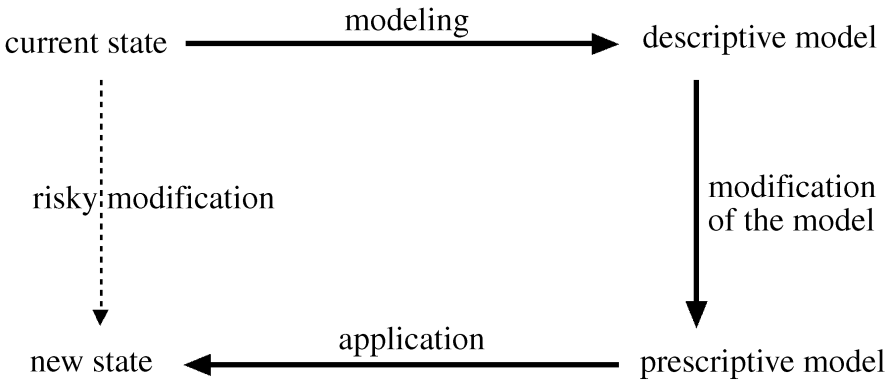



**Fig. 3.** Application of an explorative (i.e. transient) model

(descriptive) models somehow imitate the appearance of the originals. This is also true for non-material models of some objects that may be distant or non existing ("*virtual reality*").
- *Formal (mathematical) models* (e.g. the formulas used in physics and chemistry) are descriptive, too. Unlike educational models, formal models do not resemble the reality they describe. They allow us to analyse or to forecast phenomena of the real world. Other more complex models like a flight simulator often contain such mathematical models.

## 5 Models in software engineering

Models can be found in all areas and applications of software engineering. Some of them are discussed below.

While software developers create concrete models (see below), people who do research in software engineering work on notations and methods for developing such concrete models. Finite state machines and state charts, Petri nets and data flow diagrams are a few examples of models that use such notations.

### *5.1 Prescriptive models for software engineering*

Most of the models used in software engineering are prescriptive, for instance:

- *process models*, like Cleanroom Development, or Extreme Programming
- *information flow models* like the diagrams used in SADT
- *design models*, like class diagrams, or boxes representing the module structure
- *models of user interaction*, like use cases, or interaction diagrams
- *models of principles used for constructional details*, like design patterns
- *process maturity models*, like CMM or SPICE (which are actually sets of models). In these cases, each model implies a criterion for judging existing processes.

### *5.2 The document chain*

When software is developed in the traditional waterfall approach, documents are generated each of which is prescriptive for the next one. Only the requirements specification is double-sided, because it describes the user's needs, and it prescribes the product to be developed (see Fig. 4). It is this double role that makes the specification the most important software component. The chain runs from the specification to the architectural design, from there to detailed design, code, and finally execution. User's manual and test data are descriptive models of the specification; they can replace the specification for certain purposes.

One of the hard, basically still unsolved problems of software engineering is the difficulty of maintaining the logical chain of documents when any of the documents is modified. The identification and subsequent modification of other components in order to keep the whole system consistent is called *tracing*. When the tracing follows the sequence in which the documents have been (or should be) produced it is called forward tracing. When the origin of change is in one of the late documents (e.g. in the code), we need backward tracing. The problem is extremely hard because in every step of the sequence some information is lost, while some other information is added. This is why there is no automatic correction of related documents.

### *5.3 Software as a model of the world*

> *"In many systems the machine embodies a model or a simulation of some part of the world. (...) The purpose of such a model is to provide efficient and convenient access to information about the world. By capturing states and events of the world and using them to build and maintain the model we provide ourselves with a stored information asset that we can exploit later when information is needed but would be harder or more expensive to acquire directly."* (M. Jackson in [7]).

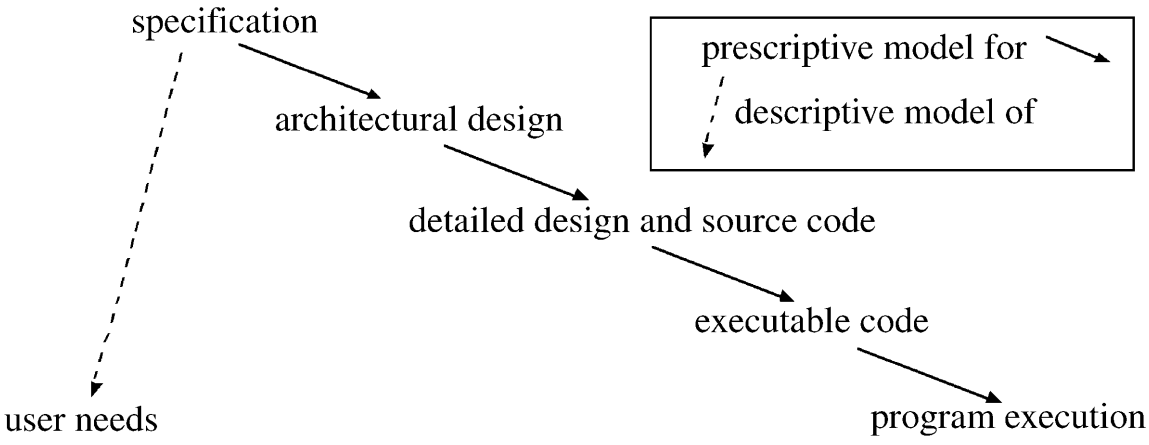



**Fig. 4.** The (simplified) document chain; some documents (like the user's manual, e.g.) and some arrows bypassing some of the documents are not shown in Fig. 4

One of the popular statements in software engineering says that many (or even all) software systems represent some part of the world. Is that true, or is it a mere conjecture?

Traditional (i.e. electrical and mechanical) engineers usually do not create models; their artefacts are tools that do not model anything. Neither a bicycle nor a radio or a house is a model of something; they are means that enable us to satisfy our needs better, and with less effort than without them. And many achievements can only be thought of when technical solutions are available: nobody could possibly walk to the moon, or watch bacteria at work. Technique enables and provides possibilities, usually without models. The symbol of engineering, the wheel, is an invention that does not model anything.

While software systems in general serve the same purpose as other technical solutions, i.e. help us in achieving something, they are not only described in terms of the real world, but they often actually represent it. Explaining a software system, we tend to say things like "In this module, the customers are stored, and this procedure will decide whether or not their orders are accepted." The computer system seems to be a doll's house, a mirror of the world outside the computer.

Why is that so? What is so special about our systems?

Our emphasis on models has good reasons:

From the very beginning, man as a thinking and deciding creature with a large memory has been the ideal for general-purpose computers. Computers (and hence computer software) imitate man. Centuries ago, people dreamed of machines that could generate music and play chess, and a chess playing machine was in the mind of Konrad Zuse as early as 1937, when he built his first computer. (He even expected that fifty years later, probably a machine would play better chess than a human; that was very close to reality!) That means: computers were designed to mimic (i.e. model) human thinking, even though we do not really understand how our brain actually works. Most of the tasks we give to the computers have traditionally been human tasks, and we simply use the models we have traditionally in mind. There have e.g. been accounting systems ages ago, and there is no reason to change our models when a machine does the job.

When we produce new software, our fantasy and creativity are insufficient for creating something that is *really* new, because we are bound to the world we know, and we cannot invent something that we cannot imagine. The world of software is completely artificial, and it is for our soul as comfortable as a box made from stainless steel would be for a bird. Therefore, we *need* to rely on notions and ideas from outside the box, i.e. on models of the (mostly physical) world we live in. In object oriented programming, that approach was made one of the fundamental concepts, but it is not that new: in JSD (Jackson System Development [1]), we start by modelling the real world, before we introduce mechanisms which can later be implemented by machines.

Modelling the world in corresponding structures has also another advantage that Jackson mentioned in the seventies already: since the world is subject to change, the software must be changed too. Changes in the world are usually consistent with the existing structures. When the laws for the tax on cars are modified, some software systems used for computing that tax have to be modified as well. If entities like cars and owners exist in the software, such changes are comparatively easy. If they do not exist (say there is only an entity "taxable object"), then the changes are very hard to implement.

However, the fact that we only build software systems that imitate the existing world severely limits our possibilities. It is precisely the departure from existing models that marks the breakthroughs that many engineers have achieved. In the early days of combustion engines, cars looked like carriages; but that turned out to be a useless limitation. Using high frequency radio waves for communication is very different from any traditional way of communicating; that is why it is far more powerful. In some limited areas, the same principle has been applied

to software systems. Quicksort and Heapsort, two sorting algorithms whose performance is far better than that of traditional sorting algorithms, are based on principles that nobody would apply when sorting playing cards or papers in a file. Very few people can actually understand *how* a Fast Fourier Transformation works, but everybody is surprised to see *how well* it works. In all of these examples, the problem to be solved can easily be described in mathematical notation, without any fuzziness and ambiguities, allowing for the application of formal methods. It is hard to say whether or not we will some day be able to find similar approaches for problems that are really complex and hard.

When the first power lines were built, engineers would try to have a straight cable from source to sink in order to avoid losses due to frequent changes of the direction of flow. They apparently had in mind some flow of material. That model was no longer necessary when the theory of electricity was sufficiently well understood. In software engineering, we are not nearly at that point.

### *5.4 Metaphors for users and developers*

> *Each XP software project is guided by a single overarching metaphor. (...) Sometimes the metaphor needs a little explanation, like saying the computer should appear as a desktop, or that pension calculation is like a spreadsheet. (...) As development proceeds and the metaphor matures, the whole team will find new inspiration from examining the metaphor. (...) The metaphor in XP replaces much of what other people call "architecture."* Kent Beck [3, p. 56]

In XP (Extreme Programming), metaphors play an important role. Kent Beck seems to be sure that software architecture and user interfaces are congruent. If it looks like a desktop (from a user's perspective), the developer should think of it as a desktop, too.

This concept oversimplifies the situation. The dustbin on the screen is not architecture; it is a simple representation for a highly complicated device. The metaphor is weak: it simply represents the fact that we can get rid of something by moving it into the dust bin, and we can retrieve it as long as the bin has not been cleared. Everything else is wrong: the electronic bin is often empty, but never full. Any file, even a very large one, fits into the bin. The bin contains files that are well organized, and do not have any influence on each other: files retrieved from garbage do not smell.

Users, in particular beginners, need metaphors in order to master their difficulties. In many cases, the metaphor is the only explanation the user gets. (Did anybody ever read the specs of a desktop dustbin?) The metaphor should be "watertight", i.e. the system should not corrupt it by inconsistencies. The happy few who use Macintosh computers must move a disc into the garbage not in order to destroy it but in order to release it from the computer. (This is not nearly as bad as the fact that the unlucky many have to click on "start" in order to shut down their systems, which resembles the idea to switch off a motor bike by using its kick starter.) When I send a letter, I want it to be delivered in due time and in good shape; if the address is wrong, the letter should return to me. These expectations hold for electronic mails as well. But sometimes we receive an error message, though the message *did* arrive at its destination.

A model for the user does not imply any model for the software architecture. In most cases, the metaphor is faked, i.e. its outside appearance is not at all similar to its inside structure. The architect can use the metaphor as a specification (of the outside appearance), but not as a hint for the architecture. When an engineer designs an aeroplane whose rudders are controlled electronically ("fly by wire"), there is no need to imitate any mechanical gear. The architecture is very different from an equivalent mechanical solution.

## 6 Risks from using models

> *A message to mapmakers: highways are not painted red, rivers don't have county lines running down the middle, and you can't see contour lines on a mountain.* William Kent [2]

Good models can replace the original very well. Therefore, good models tend to be confused with the original; very often the user does not even notice that there is such a thing like a model. For instance, most people believe that they can actually see a text file on the screen. They rarely think about all the problems related to the difference between a file and its representation. And they believe that all the details they see on the screen are the actual details of their files. Other, more threatening examples follow below.

### *6.1 A distorted view on the world*

> *The world is simply object oriented.*
> A professor of informatics, 1992

The French word *deformation professionelle* describes the distorted picture many people have due to their professional attitudes. A medical man tends to classify the people he knows according to their diseases, sometimes even in private life. A teacher of logic tells his students that any statement is either true or false; if he applies this wisdom to the education of his children, he will probably fail. And many software people are so happy about the power of object-oriented programming that they really believe in an object-oriented world. When they are hit by counterexamples, they will not hesitate to explain to us that the reality is wrong, but their view is right.

That is ridiculous. One of the most important achievements of modern (i.e. 16th century) physics was the perception that we know nothing but models. In order to improve our models (i.e. in order to make them more useful), we have to compare the reality and our models again and again, and if they do not agree, the reality is always right and the model is always wrong. Those who believe in their models are not scientists but missionaries. Software engineering circles are full of them.

### *6.2 Examples from everyday life*

The effects described above are frequently found, but usually less obvious. Software people call this "garbage in, garbage out".

If an analyst compiles the requirements in a requirements specification, her result is, as mentioned above, a model of the requirements she was given. Very often, the people she talked to were not extremely competent, and they were not able to check the written document against their real expectations. The developer will nevertheless take this document as the real and complete collection of requirements.

Words are magical: when we count branches, i.e. the if-statements in a piece of code, we get a number. There is nothing wrong with this number. But if somebody calls this number "the complexity of the program", he will usually forget very soon the primitive background of this measure, and will accept the number as a model of the complexity.

When people have tested a program for a while, identifying $n$ bugs in the code, they tend to believe that they have found all the bugs. And they tell us that the program contained $n$ bugs before testing. (Which implies the good news: now the program no longer contains any bugs.)

We all know: the program does contain more bugs. But there is at least a fair chance to find any particular bug by testing. Other deficiencies, like poor readability of the code, bad structure, lack of useful comments etc., cannot even be discovered by testing. When the results of the system test are accepted as the only indicators of software quality, the developers will inevitably try to optimise their work for this particular goal, producing unmaintainable code rather than sound software.

We can put the same statement in a more constructive way: in order to improve the situation in software engineering to achieve more reliable, robust, efficient, and maintainable software, we must develop models of software quality which reflect those properties. Next, we must teach the model and show its advantages. Finally, we must make sure that producing good software is not only beneficial for the organisation but also for the software developers.

Or even more down to earth:

- We need metrics far beyond those introduced by Halstead and McCabe which generate useful results that describe more than one very limited aspect.
- We should apply such metrics widely.
- We should not trust in people who insist on working without decent models, i.e. without decent plans, requirements, metrics, etc.

## 7 Descriptive versus prescriptive models in research

As stated in Sect. 2, creating models is the purpose of research. Scientists have provided lots of models since ages. Some of them (in Physics and Astronomy) date back to ancient times. All those models are descriptive.

Engineers need prescriptive models for building things, but their research produces descriptive models, too. We have excellent models of electric circuits, bridges, and mechanical devices. Software engineering seems to be an exception. Most results in this field offer new prescriptive models, like process models, or techniques for various activities. Is there any reason for this special position?

### *7.1 The difficulties of descriptive models*

In order to construct a descriptive model, we need to know the original very well. In software engineering, the original is the real world of software projects, with requirements that are neither complete nor precise, with customers who tend to change their mind, with developers who suffer from insufficient education, with existing software that is very hard to modify, to name only the worst problems. In short: the real world of software projects is a mess.

In our group at the University of Stuttgart, we have been developing a system called SESAM (Software Engineering Simulation using Animated Models) since 1990 [5, 6]. SESAM is based on the idea that it should be possible to coach software project managers in the same way aircraft pilots are coached; SESAM is not a flight simulator but a software project simulator.

This simulator depends on adequate descriptive models of software projects and software project management dynamics. In order to discuss the modeling problems and challenges we experienced with SESAM, a short introduction on SESAM is necessary.

The user or "player" (who is supposed to be female in the sequel) takes the role of a software project manager; SESAM simulates the rest (customer, documents, process, employees). The player will start from an initial setting as a new project manager. The interactions between player and project are handled via keyboard and screen. The player receives messages, and enters her commands in order to make her project proceed. She can hire or fire employees and ask them to perform any of the tasks that are useful for software development (like start preparing a specification or revise the design document).

On the other hand, she receives messages about the things that happen in her project (for example, when

documents are completed or when an employee leaves the project). The time scale is compressed in order to cover a whole project in a couple of hours. When the game is over, the player receives her score, and some detailed analysis of her performance.

SESAM uses two models, one for the state of the project, the other one for the rules and relations that apply to software projects in general (Fig. 5). When running a game, the game state (which models the situation in one particular software project) is subject to continuous modifications. The rules and relations that determine these modifications constitute the so-called *project model*, which can be further divided into a static model and a dynamic one. While the *static model* defines the types and relations from which our virtual projects can be built, the *dynamic model* contains all the rules that represent the invisible mechanisms of a software project. Typical information contained in the static model is the set of document types that are developed in the project, and the fact that a programmer may read or write a document. A dynamic rule describes changes, like the effect of a review on the document that has been reviewed, or the effect of a meeting on the participants.

- The tutor must define the initial state of the project. This definition is transformed into an internal representation, the so-called *game state*. As simulated time proceeds, the game state (which models the situation in a real software project) is subject to continuous modifications.
- The rules and relations that determine these modifications constitute the so-called *project model* (or simply *model*), which can be further divided into the *static model* and the *dynamic model*. While the static model defines the types and relations from which our virtual projects can be built, the dynamic model contains all the rules that represent the invisible mechanisms of a software project. For better efficiency, the dynamic model is transformed into an internal representation (the executable model), which is interpreted by the simulator.

Typical information contained in the static model is the set of document types that are developed in the project, and the fact that a programmer may read or write a document. A dynamic rule describes changes, like the effect of a review on the document that has been reviewed, or the effect of a meeting on the participants. In the game state, the actual size of all documents and the number of errors in those documents is recorded, as well as the current motivation of the developers.

While developing and using SESAM we experienced typical problems arising with descriptive models.

The project model represents a theory of software projects that is based on empirical data. Such data is extremely hard to find; little has been published, and investigating such data in industry is practically impossible because those data have rarely been collected. Therefore, the model must be validated. But validation suffers from similar problems. There are no data to compare with. For the largest model so far [4], this validation has been done, though with a huge effort.

Other problems remain. Our model covers only a small fraction of the world; it is a compromise of simplicity and realism. Is it acceptable to ignore the private life of software developers? Which attributes of documents need to be modelled? Is it sufficient to count errors, or do we need different classes of errors?

And, last but not least, our models are not only limited due to technical reasons or in-sufficient knowledge; we had to learn that our students couldn't handle very complicated models. Therefore, we no longer try to

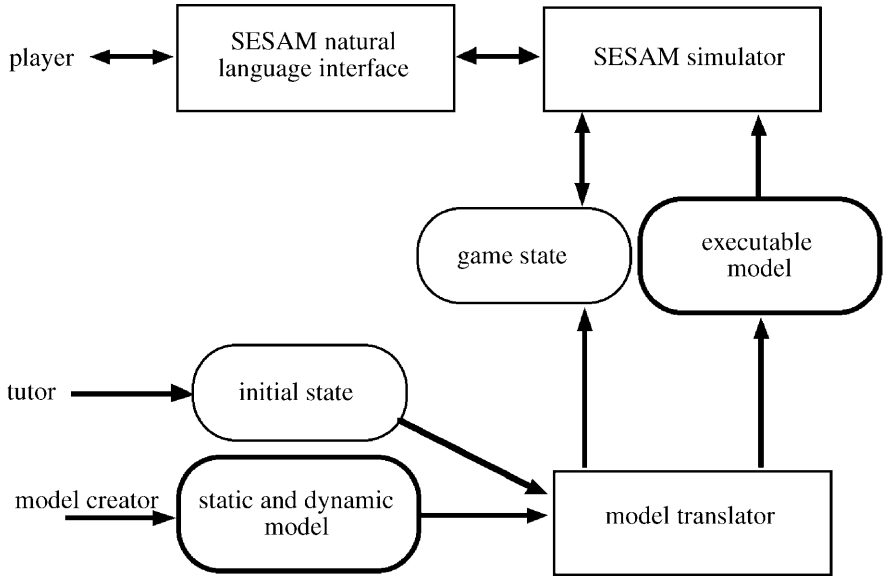



**Fig. 5.** SESAM architecture, simplified; models are shown in rounded boxes

add more detail to the models. Students should have a fair chance to succeed, because success is a reliable and cheap motivator. A good model is sufficiently detailed to be interesting, but sufficiently simple to allow for good results without weeks of training.

### *7.2 The dangerous charm of prescriptive models*

While a descriptive model like SESAM has to represent its original very well (even when the player behaves stupidly), a prescriptive model can command whatever its author prefers. Their authors tend to take a position like "we do not care in what mess you are, we prefer to tell you what a wonderful world you could possibly live in."

That position seems to be reasonable provided the better world does really exist, and so does a path from the presence to that paradise. But most research projects have no chance at all to get to the point where they could possibly demonstrate that their ideas are applicable, or even superior. Most of them terminate when they have produced a doctoral dissertation and some printed paper, and their influence on the outside world remains very limited.

The problem here is not the fact that many of the ideas and concepts may be of little value; research has to produce a lot of garbage in order to yield a few excellent results. The problem is that most ideas are never tested. Scientific magazines, conferences, funding organizations, and university departments support the breeding of new ideas, but not their careful and time consuming evaluation and improvement.

Maybe there are too many models lacking the maturity that results from steady research. If more people contribute to our understanding of the existing world by analytical and empirical work, and evaluate what has been around for a while, we will get more useful results, and we will experience a steady increase of our knowledge, as expressed in generally accepted models.

But that requires a change of goals in research: Those who do not present new ideas, but compare and improve existing ones, should receive far more recognition.

## 8 Summary

In this article, I have hopefully demonstrated:

- Models are very important, in particular for software engineers.
- Like other central notions (e.g. information), the term "model" is hard to define.
- People using models should make sure that they do not confuse their models with the reality. And they should draw conclusions from their models only very carefully, taking into account the limitations of the models.
- Creating prescriptive models is easy, but greatly improved descriptive models are what we desperately need.

*Acknowledgements.* Three anonymous reviewers have contributed many useful objections and corrections. Michael Jackson in England has volunteered to read and comment my manuscript. Finally, Martin Glinz, the guest editor, helped to polish the final version. Many thanks to all of them for their time and effort!

## References

1. Cameron, JR. (1986) An overview of JSD. IEEE Trans. on Softw. Eng. SE-12(2): 222–240
2. Kent, W. (2000) Data and Reality. North-Holland. Republished by 1stBooks Library (including an electronic version, see `http://www.1stbooks.com/`), April
3. Beck, K. (1999) Extreme Programming. Addison-Wesley, Reading, Mass.
4. Drappa, A. (2000) Quantitative Modellierung von Softwareprojekten. Dissertation, University of Stuttgart. Shaker-Verlag Aachen
5. Drappa, A., Ludewig, J. (2000) Simulation in Software Engineering Training. 22nd ICSE, Limerick, pp. 199–208
6. Georgescu, A. (2003) Web pages on SESAM, see `http://www.informatik.uni-stuttgart.de/ifi/se/research/sesam`
7. Jackson, M. (1995) The world and the machine. Proc. of the 17th ICSE, Seattle. ACM, pp. 283–292
8. Stachowiak, H. (1973) Allgemeine Modelltheorie. Springer-Verlag, Wien etc.

# Requirements Elicitation Techniques

Dr. Ruzanna Chitchyan

Based on book “Discovering Requirements” by I. Alexander and L. Beus-Dukic

# Overview

- Requirements Elicitation Techniques
  - Interviews
  - Observations
  - Group Elicitation/Workshops
  - Prototyping
    - Use Cases

# Interviews


- **Types:**
  - Structured: provide a set of questions
  - Open-ended: provide some questions and follow on with context-related new questions
- **Number of Interviewers**
  - Single vs. team
    - Listen
    - Think
    - Write (notes, models, sketches, processes, use cases…)
    - Ask follow up questions
- **Number of Interviewees**
  - Single vs. few
  - No boss in the group

# Interviews

- **Advantages**
  - Engaging with stakeholders: direct attention
    - Individuals get attention
    - Their input becomes part of requirements
    - Requirements are translated into products/services
  - Dialogue and feedback
    - Immediate check of your understanding with interviewee
    - Obtain further feedback/clarifications
  - Follow up interview
    - Check understanding
    - Fill in gaps
    - Ask additional questions
- **Disadvantages**
  - Interviewees state only what they know:
    - “faster horses” vs. cars (not for innovative solutions)
    - Tacit knowledge: cannot tell, but could show (e.g., riding a bike)
  - Captures only one point of view at a time:
    - a lot of qualitative data that is hard to analyze
    - Interviewees cannot hear each other’s views/resolve conflicts

# Interviews: How to

- **Plan:**
  - Plan in advance:
    - Find out about the interviewees
    - What questions will you ask
      - What, why, when, where, how and who?
    - Any similar systems/sketches/drawings
    - Start with easy questions, keep open-ended ones for the end
- **Allow flexible departure from plan**
  - Follow relevant leads
- **Record/take notes**
  - With permission
  - Switch off controlled by interviewee
- **Validate findings**
  - Check you understanding (e.g., feedback to summary, or re-visiting tricky points)
  - Validate notes by using these as input on other interviews
  - Validate notes with the interviewee after fact

# Interview Extract

***How much electricity/gas do you use?***

RC: Do you know how much energy do you use?
KS: What do you mean, in terms of price ?
RC: Well how do you approach that? How do you perceive it? Because we want to understand what people think, how do they appreciate how much energy they use.
KS: Well, I probably think in terms of how much of gas and electric that I use, really. So I do try and be conscious of that, yes.
RC: In what terms do you think about gas and electricity?
KS: I think in terms of price, yes. And in terms of trying not use so much environment.
RC: But do you know how much you pay?
KS: Yes.
RC: Are you expecting a certain amount every month?
KS: yes, I pay £70 a months, which I think is too much, so I try and, you know, use … more trying to look for ways to use less. I pay £70 a months gas and electric.

# Observation


*"It is better to see once than to hear 10 times"* e.g.: air traffic controllers work in a busy environment with lots of data and many people collaborating in fast, specialized language.

- Observer "immerses" into observed environment for some time along with the observed subjects

- **Types:**
  - Silent observation
  - Talked through observation
  - Apprentice
  - Also: Overt vs Covert
- **Advantages**
  - First-hand experience of context (what will/not work, how does it feel)
  - Reveals issues that no other technique can discover
- **Disadvantages**
  - Can be time consuming
  - Collected rich data can be hard to make sense of
  - Not inactive on effects of newly induced software
  - Danger of "going native"

# Observation: How to

- **Plan**
  - Choose the observed subjects, time, and place
  - Obtain written permission
  - Prepare for recording/documenting
  - Explain observation (if open) to the subjects
- **Observe**
  - Observe in natural setting
  - No interference with observed subject carrying out his tasks
- **Record/take notes**
  - Notes, audio, video, photos, etc.
- **Validate findings**
  - Where possible, validate findings with those observed (e.g., do I understand that in this task you did X because of Y?)

# Observation: Further Notes

- **Objectivity**
  - Not looking for objective truth, but local behaviours, values, structures, interactions
  - Focus is on subjective/local
    - Requirements and their validation are thus subjective and contextual
- **Utility**
  - "Simplified" observations often used in software evaluation
    - Measurement of bodily functions - e.g. heartbeat, respiration
    - Studies of non-verbal interactions (e.g. gestures, gaze)
    - Video analysis of prototype use
  - Can also be used for social analysis, e.g., if we intend to automate some functions, use:
    - Time-motion study: noting people's location at each given time/stage of the task they are doing.
    - Communication audit: observe who talks to whom and what about

# Group Elicitation

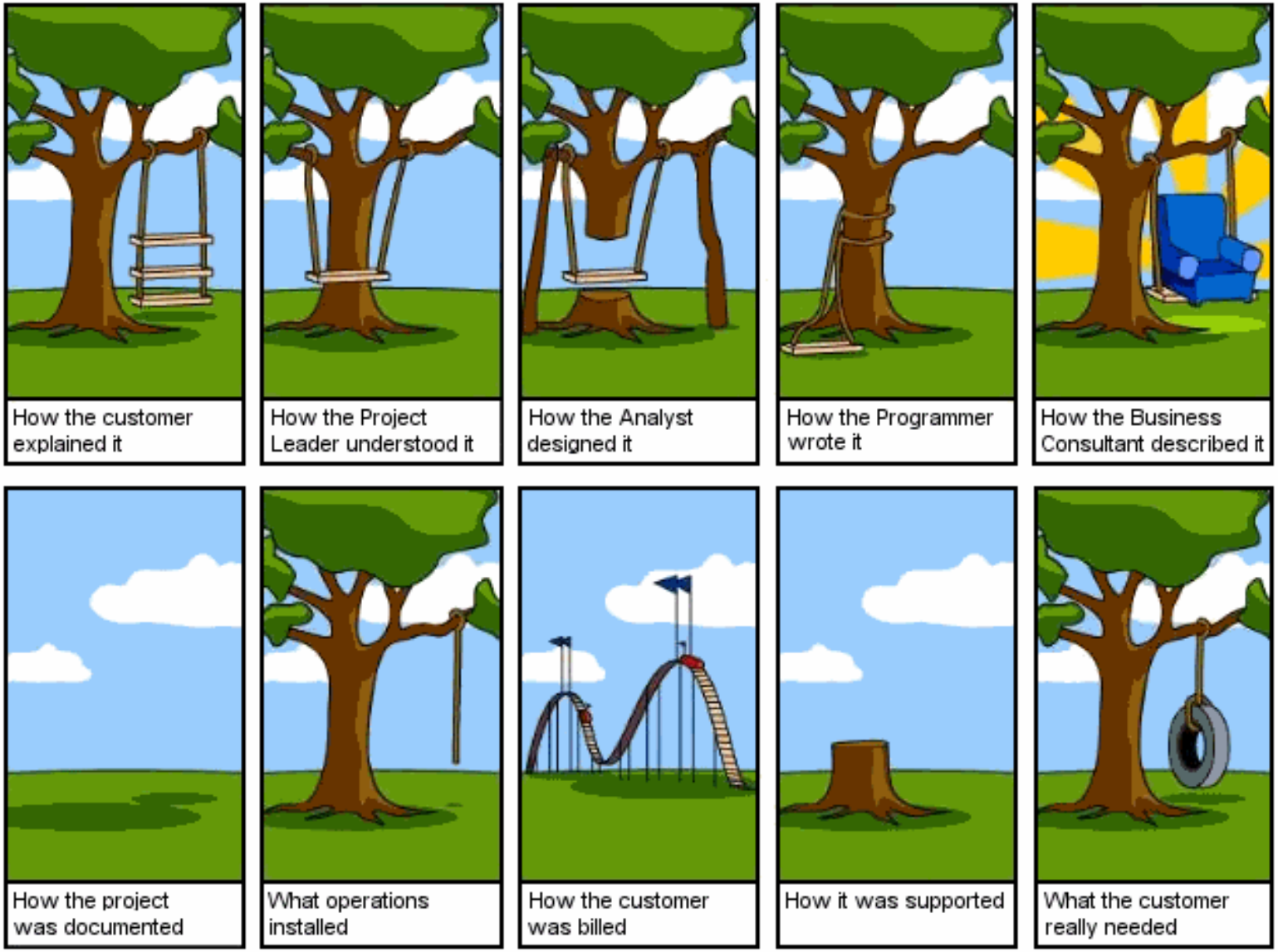
- **Types:**
  - Focus Groups
  - Brainstorming/workshops
- **Advantages**
  - Bring a several sources of information together
  - Refine collective understanding
  - Identify and resolve conflicts/inconsistencies
  - Reach agreement
- **Disadvantages**
  - May not be a “Balanced Sample” of representatives
  - Participant Discomfort (e.g., agree with boss)
  - Danger of loss of focus or joint poor decision
  - Dependency of facilitator’s skills

# Group Elicitation: How to

- **Plan**
  - Define the workshop aim (e.g., create scenarios for a given product)
  - Select invitees: include only and all relevant roles
  - Set agenda with time per each activity
  - Set up and check the room and *equipment*
- **Consider options for departure from plan**
- **Rehearse**
  - Try out group activities
  - Think through activities' sequence and relevance
- **Record**
  - Notes, sketches, or even video
- **Validate**
  - Get participants to review notes/reprot

# Why to do Prototyping?

Use prototypes in interviews and workshops to discuss with users



# Prototyping

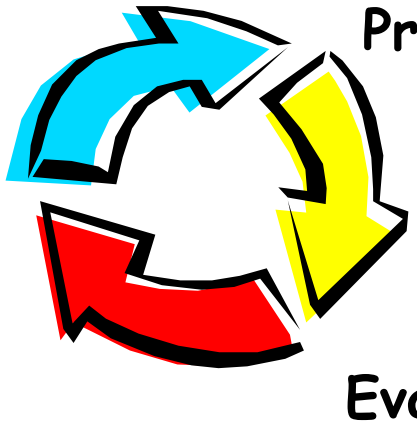
"Users don't know what they want until you show it to them" Ken Beck.

RE prototypes – artifacts used to facilitate elicitation of requirements

- **Advantages**
  - Easy and cheap to create
  - Induces feedback due to visual and/or useable nature
  - An abstract requirements can be "placed" in context
- **Disadvantage**
  - Could give impression that system is already designed
  - Could set out unrealistic expectations

# Prototyping: How To

1. Choose Users
2. Select Tasks (e.g., use cases)
3. Draft a Design
4. Check the Design



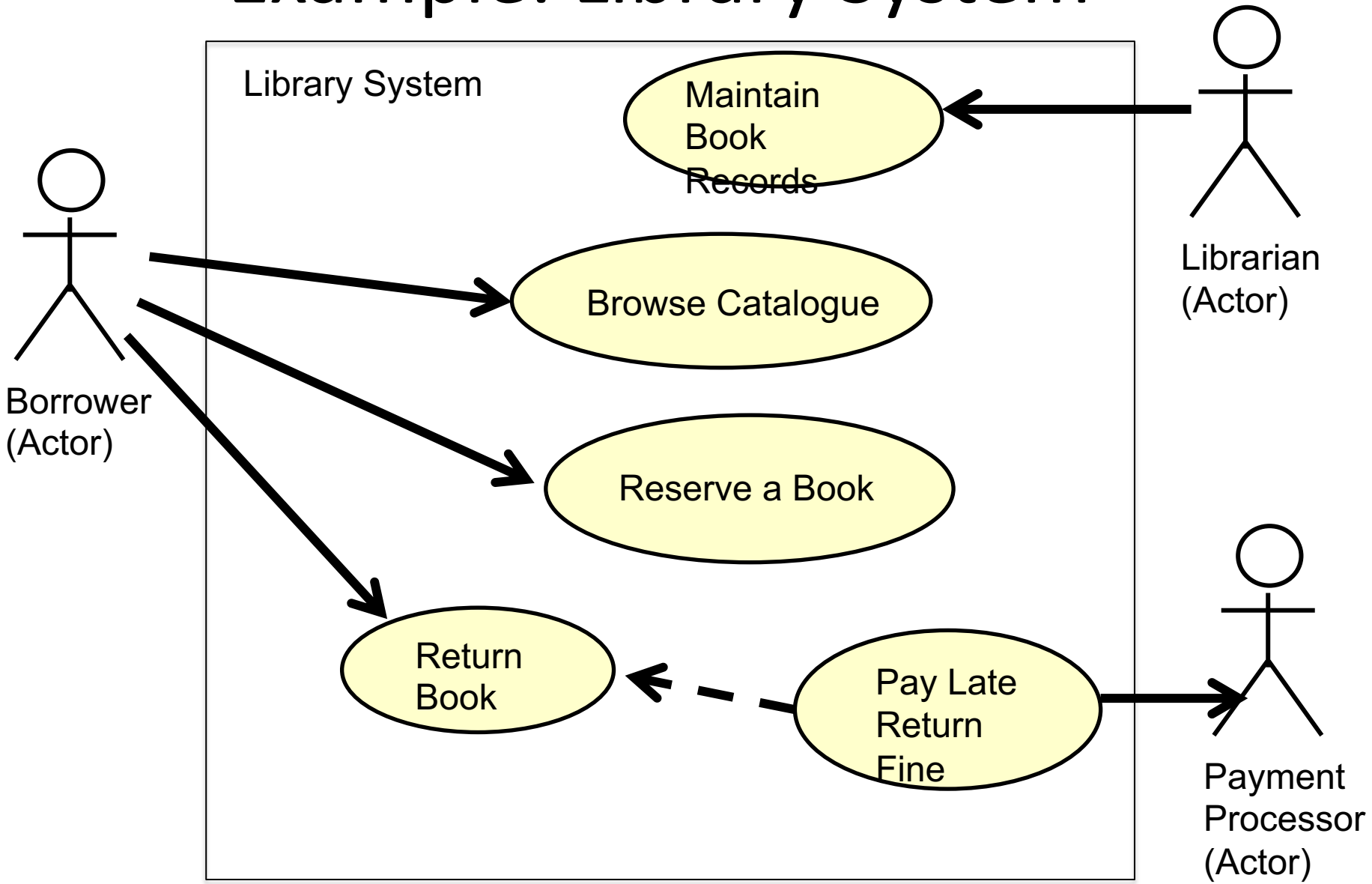

5. Prototype

6. Demonstrate
    - 6.1. Evaluate with different stakeholders
    - 6.2. Record Feedback
7. Iterate

# Use Cases

- A use case captures functional requirements of a system:
  - what a system should do in response to an action
  - Who are the actors for a given functionality
- Use cases are used for
  - Identification of Functional Requirements
  - Communication with Others
  - Testing System

# Example: Library System



# Example Use Case Description

**Use Case: Browse Catalogue**

**Basic Flow**

The use case begins when a customer wants to browse the catalogue.

1. The customer selects view catalogue option,
2. The system displays the set of listed categories (such as author, keyword, title, subject area) and a text box.
3. The customer inputs the search text and chooses a category.
4. The customer presses Search button.
5. The catalogue displays the list of all available items under the selected category.
6. …

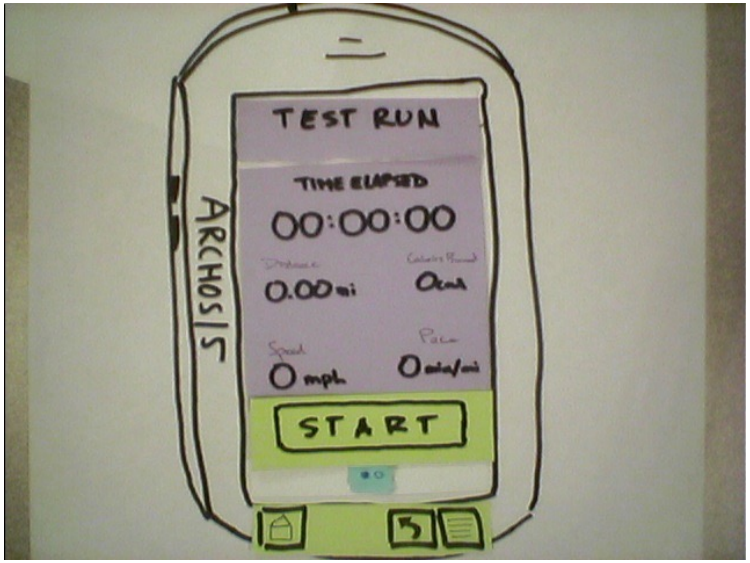
**Alternative Flows**

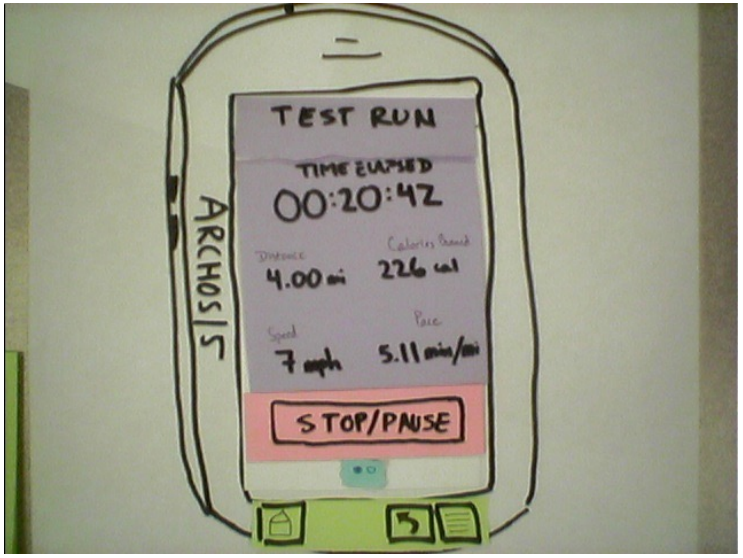
No items available,  or connection to the library catalogue fails….

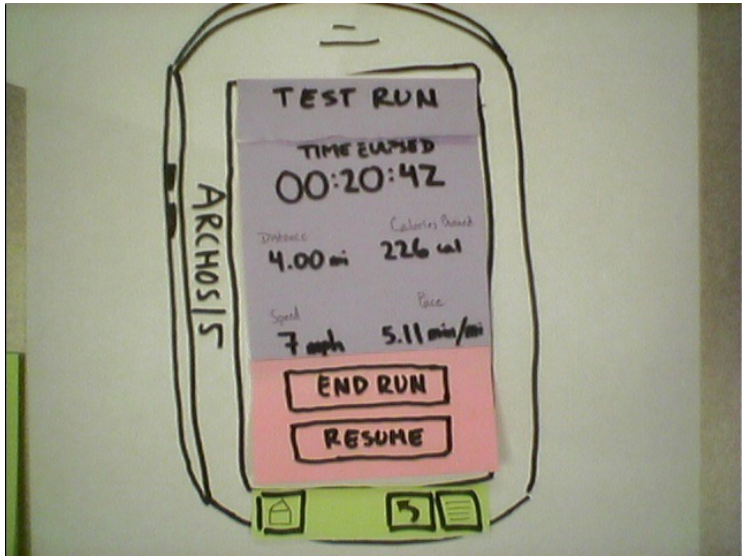
# Example Test Case Definition (Black Box)

| Test Case for Browse catalogue: Inputs | Expected Output | Observed Output |
|---|---|---|
| Input Text “Philosopher’s Stone”<br>category “keyword” | Harry Potter and the Philosopher’s Stone | |
| Input Text “J. K. Rowling”<br>Category “author” | Harry Potter and the Philosopher’s Stone | |
| … | | |

# Example: Running App

# Evaluation: Against Use Case Tests

| Test Case for Browse catalogue: Inputs | Expected Output | Observed Output |
|---|---|---|
| Input Text “Philosopher’s Stone”<br>category “keyword” | Harry Potter and the Philosopher’s Stone | Harry Potter and the Philosopher’s Stone |
| Input Text “J. K. Rowling”<br>Category “author” | Harry Potter and the Philosopher’s Stone | “no such author” |
| … | | |

# Conclusions

- **Use a combination of RE techniques**
  - Interview or observation for initial requirements gathering
  - Prototyping for confirmation and elaboration of requirements
  - Observation for evaluation of prototypes
- **Other techniques**
  - Participatory Design: users directly involved with each stage of the project
  - Joint Rapid Application Development: a multi-day workshop with 3-5 stakeholder participants working as a team
  - Focus groups: experts debating/discussing a problem…
- Each Technique has its own advantages and disadvantages: **no single perfect choice for all cases**

CHAPTER TWO

# Stakeholders

*The two most important parts of a computing system are the users and their data, in that order.*
Neville Holmes

| ***Requirement Elements*** / ***Discovery Contexts*** | **Stakeholders** | Goals | Context, Interfaces, Scope | Scenarios | Qualities and Constraints | Rationale and Assumptions | Definitions | Measurements | Priorities |
|---|---|---|---|---|---|---|---|---|---|
| Introduction | | | | | | | | | |
| From Individuals | | | | | | | | | |
| From Groups | | | | | | | | | |
| From Things | | | | | | | | | |
| Trade-Offs | | | | | | | | | |
| Putting it all Together | | | | | | | | | |

**Answering the questions:**

- Who has a valid interest in this product or service?
- How do you engage with and manage those people?
- Which requirements come from which stakeholders?

. . . so you know who you need to deal with on the project.

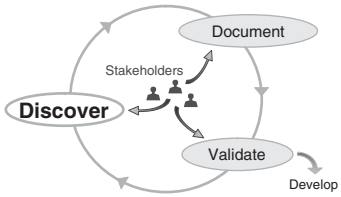
## 2.1 Summary

This chapter looks in turn at what stakeholders are, how to discover the stakeholders on your project, how to engage with them and how to manage them to ensure success.

Stakeholders are far more diverse than simply 'developers' and 'users'. A better understanding of stakeholders as beneficiaries, operators, interfacing and negative stakeholders, regulators and others, contributes to more effective discovery of requirements.

The chapter ends with a look ahead (to the rest of the book) at how you will work with stakeholders to discover requirements of different kinds.

## 2.2 Discovering Stakeholders




Requirements ultimately begin and end with people – stakeholders. Constructing an onion model (Alexander 2005) [1] is a good way to start to understand and document the stakeholder structure of your project (Figure 2.1). Another starting point may be an organisation chart, as long as you don't overlook stakeholders outside the organisation.

The rings of the 'onion' are centred on the product or service itself (as it is planned to be), not on the project and its team of developers.

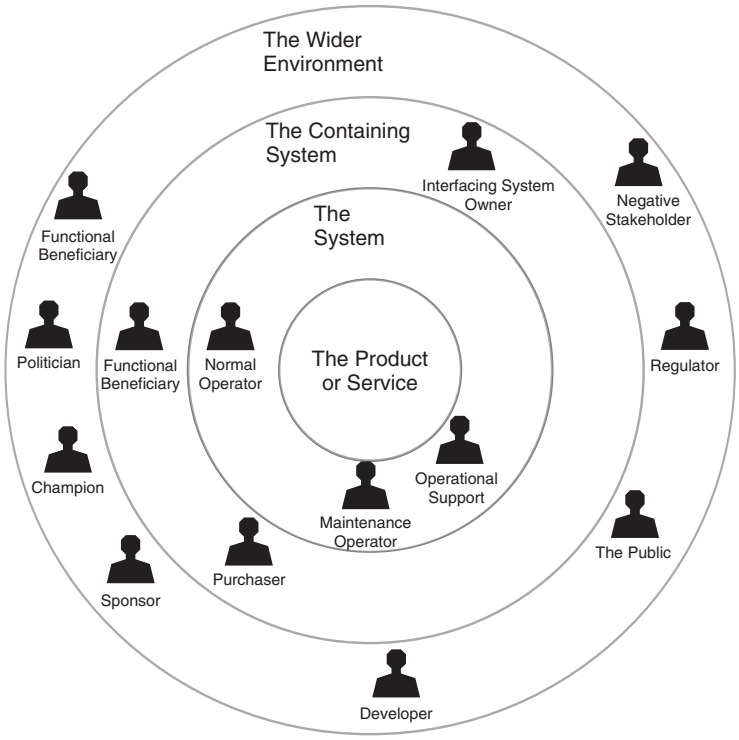



**Figure 2.1:** 'Onion' model of stakeholders in a typical system.

**Developers: Inside or Outside the System?**

Developers are very close to the product while it is under development but may have little to do with it once it is in service unless they 'wear two hats' by also having a specific operational role, such as maintenance or helpdesk (operational support). If the onion is drawn for the product when it is in service, the developer therefore appears in 'the wider environment'; the maintenance operator appears as part of 'the system'.

You could draw a *different* onion model for the product under development, in which case your system will be 'the development system' (your software factory, for instance). In that case, the developer will be inside, and operators outside. To a degree, we use onion models to counteract the tendency to marginalise operators and other stakeholders, so we tend not to focus too much on the development environment at this point.

**Table 2.1:** Operational roles within 'the system'.

| Generic role | Work done by role | Example |
|---|---|---|
| Normal operator | Interacts with the product to deliver results (to functional beneficiaries) | Tram driver drives the tram |
| Maintenance operator | Services and repairs the product (i.e. carries out both planned and unplanned maintenance) | Mechanic services the tram |
| Support operator | Provides help and co-ordination to keep the other operators productive | Roster co-ordinator allocates drivers to trams each day |

### 2.2.1 Operational Stakeholders within 'The System'

The innermost ring of the onion (around its solid centre) is 'the system'. This means the people, procedures and products that together deliver results to the world outside. Notice that 'the system' is not the same as a product, a service, a computer or a piece of software, though it may contain any of those things.

The people who form part of the system around a product or service are the operational stakeholders: they take part in day-to-day operations.

Typical systems contain several operational roles[1] (Table 2.1). In a complex system, such as a warship or a manufacturing plant, there may be hundreds of different operational roles.

### 2.2.2 Stakeholders in the Containing System and Wider Environment

#### *Beneficiaries: Who's it for?*

*The man who pays the piper calls the tune.*
Traditional English Proverb

Where the onion model in Figure 2.1 gives a product-centric view, Figure 2.2 offers an alternative, project-centric view, which may seem more familiar. This is because projects have traditionally thought of the world as consisting of two groups: themselves and 'users', by which they meant everybody else. Figure 2.2, which was created to help people in a transport organisation think about their stakeholders, goes a little further, distinguishing beneficiaries–people who are intended to benefit in some way–from the others.

[1]You may hear the term 'actors', which is what the Unified Modeling Language (UML) calls operational roles. Note that if there is an 'intelligent' software system at the far end of an interface, UML considers that an actor as well. Essentially, UML is interested only in actively interacting roles, not in stakeholders in general. Interfaces are described in Chapter 4.

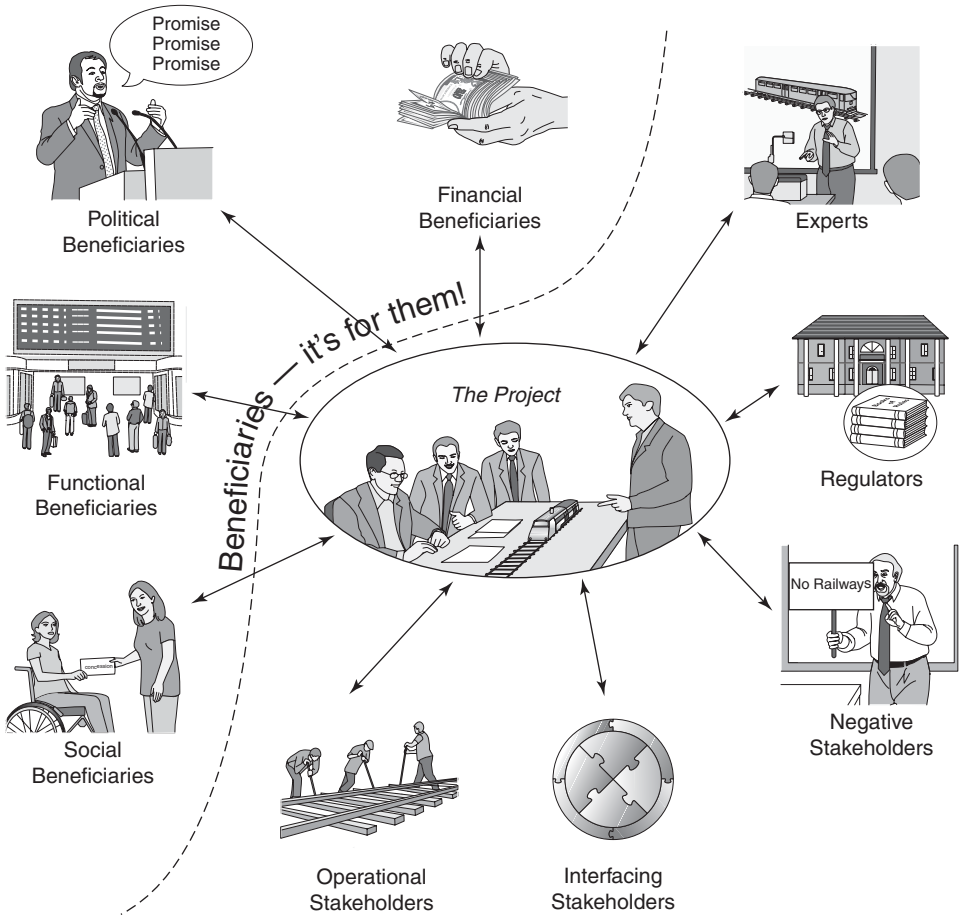



**Figure 2.2:** Beneficiaries and other stakeholders in a transport project.

A system, such as a tram, for example, forms part of some larger system or context. Systems are created to provide services or results to their beneficiaries. Beneficiaries include several very different kinds of stakeholder, but by no means all of them.

For example, the driver of a tram is not driving it for his own enjoyment. The intended beneficiaries of the tram service are the passengers, who pay to be carried along the tram's route. In fact, the service is for them.

You can answer the question 'Why are we operating this service?' by studying the benefits that it provides. Those benefits are not only to the passengers: the tram company, its directors and its shareholders benefit financially (as of course does the driver, in a small way).

We can call the passengers 'functional beneficiaries', and the shareholders (who are arguably further from the product, i.e. in the 'wider environment' ring), 'financial beneficiaries'. Projects also often bring political gains to people

whose careers benefit from a project's success. Such people may be career politicians, or may be political with a small 'p', within the corridors of your organisation.

### *Regulators*

One role that is of crucial importance to services or products for the public is that of the regulator. There are regulators of financial correctness, of food, of medicines, of the radio spectrum, of broadcasting, of aviation, of railways, of manufacturing - in fact, of nearly everything.

Many products are subject to multiple regulators. Software is in nearly everything, so it is regulated in multiple ways (see box, 'Statutes, Regulations, Standards').

**Statutes, Regulations, Standards**

In some countries, such as the UK, regulation takes several forms:

- primary legislation in the form of **statute laws** passed by Parliament: for instance, the Financial Services Authority (FSA) was created by statute law;
- secondary legislation in the form of official **regulations** issued by government departments or other regulatory bodies, such as the FSA, under authority given to them by Parliament; these are often much closer than statutes to the level at which projects work;
- **international standards** imposed by bodies such as the International Standards Organisation (ISO);
- **national standards** imposed by general standards bodies such as the British Standards Institution (BSI), or specialist professional bodies such as the Institution of Engineering and Technology (IET);
- **industry 'best practice' and 'guidance'**, which are often essentially mandatory within an industry, especially where safety is concerned; businesses may comply with these voluntarily, to give their offerings a mark of quality and hence seek a competitive advantage, or may be obliged to comply by regulation.

As if all that were not enough, companies often impose their own **company standards** on their projects, e.g. imposing a software development process.

Projects, in turn, often write their own **procedures**, for example, describing how to structure requirements and use cases in their chosen requirements tool.

Standards and regulations can be seen as **reusable sets of requirements** that are shared between all systems in a domain. Some are voluntary 'best practice' or 'guidance' (see box, 'Statutes, Regulations, Standards'); some are essentially mandatory. More is said on requirements reuse in Chapter 13.

Standards and regulations form a significant percentage of all requirements. This shows, incidentally, that not all requirements come from 'users'.

### *Interfacing Roles*

Almost every system has significant interfaces to other systems, services or businesses (whether existing or future). For example, many devices such as laptop computers and mobile phones have a Bluetooth[2] short-range radio interface, allowing them to exchange data with other Bluetooth-compatible devices.

Chapter 4, which looks at context, interfaces and scope, describes how you can analyse interfaces to identify requirements on your system. Effectively, your freedom of design is constrained by your interfaces, especially if they are already fully specified.

Identifying the systems or services that you need to interface to is obviously a vital step towards creating the right requirements and the right product. The project or organisation responsible for the other end of each interface is an important stakeholder in your project.

### *Negative Stakeholders*

Generally, your project should aim to satisfy most of your stakeholders – but that does not necessarily include negative stakeholders.

Negative stakeholders range in attitude from peaceful opposition to active hostility. They may threaten or cause harm to a project, intentionally or incidentally. Your competitors may not wish your project to succeed, but they will generally stay within the law in their opposition to it.

Peaceful and lawful opposition can include, for example, writing letters to the government and collecting signatures to oppose the construction of a road, airport, shopping centre or power station that threatens to destroy assets valued by the stakeholders. Such assets can be of any kind, for example, historic buildings, the natural environment and wildlife, and existing jobs and businesses, as well as more abstract things such as leisure, safety, peace and quiet, and privacy. As a result, negative reactions to a project can be extremely diverse.

Security threats also come in many shapes: from thieves, vandals, hackers, disgruntled employees, criminal gangs using viruses and malware, terrorists

[2]Bluetooth is currently an industry standard under the control of a Special Interest Group (SIG), a trade association. Many industry standards eventually become International Standards.

and military enemies. Thieves and hackers may be opportunistic, rather than intentionally hostile. The others are, presumably hostile by intention.

For military systems, the enemy is a key stakeholder. He may use any means to confuse, disrupt, deceive or destroy. Countering these threats leads to many of the requirements on military equipment, for qualities such as data integrity, confidentiality and survivability (see Chapter 6, Qualities and Constraints). New threats, such as electronic and cyber-warfare, are increasingly important concerns.

You might imagine that software projects do not face the risk of opposition. However, the public are becoming concerned about data protection for reasons of privacy and financial security. Online advertising techniques such as stealthy data collection have recently created a storm of protest on the web. Viruses are able to attack smartphones and PDAs as well as computers. Negative stakeholders for software can include the public, bloggers, campaign groups, hackers, virus writers and probably many others.

## *Sponsor and Champion*

*Without a sponsor your project will be shortlived.*
Suzanne Robertson [2]

In both commercial companies and public service, money is generally in shorter supply than ideas for how to spend it. Therefore, departments usually compete for resources. Any project that receives funding may be watched jealously for signs of weakness so that it may be killed off and its funds appropriated for something else.

The Champion has to be someone with enough power in the organisation to protect the project against 'political' threats (i.e. from negative stakeholders within the organisation). Mere technical success is no guarantee of safety. An effective Champion does not have to be technical, and is typically not involved in the day-to-day running of the project.

**Sponsor and Champion**

- The sponsor provides development funding for a project.
- The champion provides 'political' support for a project.
- The two roles can be combined but are often separate.

The Champion could be the department head who sponsors the project, but it's equally likely that the sponsor is another organisation or another part of

the development organisation. For example, an automobile maker could have a new secret project to develop a fuel cell car for the luxury urban market. This might be sponsored by international marketing as part of their roadmap for a future product line, but championed by the director of technology in board meetings.

### *Worked Example: Stakeholders in a Tram Service*

The agency planning a new tram service must carefully consider possible opposition from residents, pressure groups (such as wildlife organisations), businesses and local government. All of these could be either beneficiaries or negative stakeholders. The direction they choose depends both on their own attitudes and on the preparation, skill and tact of the project's management, including its handling of requirements.

Figure 2.3 illustrates a typical geographic situation for a tram project. The City Tram is to relieve traffic congestion by providing a reduced journey time from A to B. With closely spaced stops at places such as C, D, E, F, G, H, and an interchange with the railway at B, it should encourage economic growth in the city. A stop at D provides access to public transport for residents of the district north of the park, and could encourage further expansion of the tram service to the north.

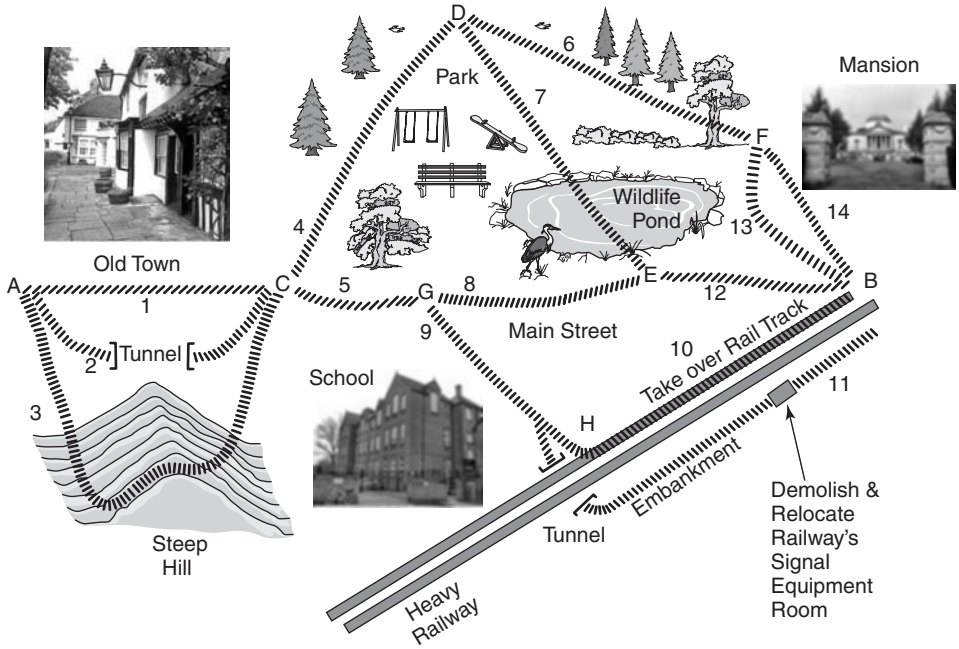



**Figure 2.3:** Tram routes: a problem in stakeholder geography.

The tram is likely to be supported by local businesses and by the developers of a new shopping mall on Main Street. Bus and heavy rail are also likely to provide support, as long as the interfaces to those services are well thought out.

However, there are obstacles. Between A and C is the old town, with an attractive townscape. A route through here could upset the town council (local government), who are otherwise likely to be in favour because of the commercial benefits a tram could bring. Local governments are powerful stakeholders, with authority over local planning issues.

To the south is a steep hill, which could cause operational difficulties for the tram in wet or icy weather. A central route is imaginable but would require a costly tunnel through the hill. Apart from the cost, a tunnel makes escape in case of fire much more difficult, so another powerful stakeholder, the railway safety authority, could cause difficulties here.

Between C and F is the town park, with scenic views, much-loved mature trees and a wildlife pond. Near F is The Mansion, a Grade I historic building protected both by law and by local and national conservation bodies; all are powerful stakeholders.

If the tram was to exit the park at E, it would avoid The Mansion but harm the wildlife pond. A partial alternative is to route the tram in a longer semicircular arc between F and B so as to avoid the street immediately in front of The Mansion.

The park could be avoided by a route through G, south of the park, but the route from G to E is along busy Main Street, threatening to worsen the traffic congestion rather than alleviating it.

An alternative to Main Street is the route southeast from G to the heavy railway at H; unfortunately, this crosses the school playground and could cause strong local opposition to the tram. From H to B, the tram could run on one of the existing railway tracks if the railway company can be persuaded to give it up, or it could run on top of the embankment south of the railway line. However, the embankment route would demand a tunnel to cross the railway, as well as the demolition and re-siting of the railway's signal equipment room, which the railway company opposes. The tunnel would raise buildability issues, as access to the railway is limited, and closures would have to be planned for nights and weekends.

The problem facing the tram project team, therefore, is to choose the best route (set of route segments linking A and B) by trading-off the constraints just described. You can see at once that each group of stakeholders (the Historic Building Society, the school's parent teacher association, the Old Town Preservation Society, the Wildlife Trust, the railway, etc) has an interest in at least one route segment, and could easily become actively hostile to the project. This kind of project has very many stakeholders (it is common to find hundreds of interested parties) and their management is a large part of the work of a successful transport project.

## 2.3 Identifying Stakeholders

There are several ways of finding out who your stakeholders are, i.e. who should be involved or consulted on your project:

- by asking your sponsor or client;
- by examining what is and what isn't shown on an organisation chart;
- with a template such as the 'onion model';
- by comparison with similar projects;
- by analysing the context of the project.

Naturally, as interviewing proceeds, you can always ask people: 'And who else should we talk to about that?'

### 2.3.1 From your Sponsor or Client

A natural place to start is your sponsor. Ask them who the other stakeholders are. Meet those people and ask them the same question, and so on. This would be an ideal approach but for one thing: project people may only lead you to people they know, within a tight group. This, is perhaps, a particular danger within a large organisation, where roles outside may be overlooked.

### 2.3.2 With a Template such as the Onion Model

The onion model or an equivalent list of typical project stakeholders can help to counteract closed-group thinking.

If you think graphically, take a copy of an onion model like the one shown in Figure 2.1.

- For each icon on the diagram, ask: 'Does our project have stakeholders with this role?'
- If the answer is yes, label that icon with the name of the role; be specific where the basic model is generic.
- Then, for each role you have identified, ask: 'Who interacts with this role?' or 'Who has an influence on this role?'
- Add new icons if necessary, and label them with their roles.
- Draw arrows between the role icons, and label these for each interaction or influence.

If you find it more helpful to use text, use the stakeholder roles listed in Figure 2.4 to identify people you need to contact, and follow this up with action.

Figure 2.4 shows a sample of typical, generic roles. Your organisation's classification may differ, as roles overlap, are lumped together or are further fragmented. In your area, people may use different names for some roles. Individual stakeholders may hold two or more roles, sometimes in surprising combinations. Expect to find a special situation on every project.

**Operational roles**
- Normal operations
- Maintenance
- Support
- Helpdesk
- Training
- Planning, scheduling
- Control, management, administration

**Beneficiaries**
- Functional beneficiary
- Social beneficiary, etc (if indirect benefits exist)
- Financial beneficiary
- Political beneficiary (not only professional politicians)

**Interfacing roles**
- Owner of interfacing system (sharing data, etc)
- Neighbouring business (mutual benefit)

  Interoperating service (sharing facilities/equipment)

**Surrogate roles** (representing or working on behalf of others)
- Champion
- Purchaser (roles here vary widely and often overlap)
  - Internal customer
  - Procurement
  - Project sponsor
  - Marketing (representing the consumer)
  - Product manager
- Developer (many roles possible here)
  - Requirements analyst
  - Designer
  - User interface designer
  - Programmer
  - Tester (very useful in requirements work)
  - Technical writer
- Manufacturer/subcontractor/supplier
- Expert/consultant (many more roles possible here)
  - Human factors/ergonomics
  - Safety engineer
  - Security engineer
  - Simulation/modelling expert
  - Legal opinion
  - Translator, cultural advisor, etc
- Regulator
  - Parliament, statute law
  - Government departments, regulations
  - Statutory regulators
  - Standards bodies (national and international)
  - Voluntary/industry regulators

**Figure 2.4:** Template: possible stakeholder roles on projects.

**Hybrid roles**

- User (= Normal Operator + Functional Beneficiary)
- Consumer (= Mass-market Purchaser + User)
- Service Provider (= Operational Support + Maintenance + Helpdesk, and sometimes Developer, Subcontractor, Manufacturer)
- Risk and Revenue-Sharing Partner (= Developer + Manufacturer + Financial Beneficiary), etc

**Negative/hostile roles**

- Vandal, graffiti artist
- Thief
- Hacker, virus writer
- Competitor
- Industrial espionage (via malware, etc)
- Fraudster
- Disgruntled employee
- Trades union
- Political opponent
- The public, residents' association, etc
- Activist, environmental pressure group, etc
- Military enemy, terrorist

**Figure 2.4:** *(Continued)*

### Surrogate Roles

You will find stakeholder roles much easier to unpick once you understand one common situation: many roles, including every kind of paid work, are surrogates, i.e. on behalf of somebody else.

For example: a company director represents the shareholders; a politician represents the public; a lawyer represents a plaintiff; a requirements engineer represents a project's stakeholders; a user interface designer represents a product's human operators.

The breadth of this list of examples indicates that surrogacy is the usual case on development projects. The old dogma, 'Go and get the requirements from the users directly', is well-intentioned, but wrong for several reasons:

- There isn't one uniform group of people who use a product.
- You don't know what using the product will be like until you've designed it!
- 'The users don't know what they want until you show it to them.' Products create requirements, as much as the other way round.
- You often can't talk to users directly but you must deal with surrogates, including yourself and your project colleagues.

**Surrogacy is valuable** because it enables a project to deal with a manageable number of people, and because it means that external parties are represented. This is necessary for several reasons:

- Large and complex projects can take many years, making it impossible to talk to 'the users'. If the aircraft carrier that is being planned today will not sail for another 25 years, many of its crew will not yet have been born. The future military and political environment cannot be fully known either. Planners and modellers are expert in working out the range of likely future 'worlds'.
- Government and its appointed regulators represent the law and the public – the voice of the people. It isn't possible to speak to millions of people, but the officially-appointed regulator is empowered to speak for them.
- A mass-market product like a car or music player will have millions of users. Their opinions can be surveyed, or a sample can be involved (e.g. to comment on a prototype). Marketing and product management professionals have specific expertise in identifying and predicting what the consuming public want.

**Surrogacy is dangerous** because people can be wrong about the needs and wishes of the people they claim to represent. (That is why old timers tell you to go and talk to 'the users'.)

- Marketing and product management have to live with conflicting interests; they represent the views of the market to the company, but also serve the company.

The best surrogacy efficiently condenses many vague feelings and intentions into a clear, representative statement. You will have to use surrogates on your project – you are one yourself. Your task in discovering other people's requirements is to serve those people as faithfully as possible. A measure of scepticism and self-knowledge is needed.

### 2.3.3 By Comparison with Similar Projects

If you have access to people who have experience of similar projects, ask them for their list of stakeholders, or get them to look for gaps in your list.

### 2.3.4 By Analysing Context

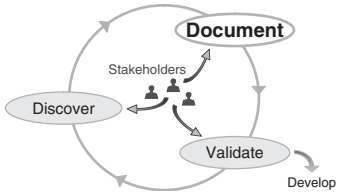
Another approach is to explore the context (see Chapter 4) of your project. For example, if your context is geographical (as with the tram service), then

searching a map of the area immediately around your tram route may reveal neighbouring businesses and services. Or again, if your context is an existing technical or service domain (e.g. hospitals), then analysing the roles and responsibilities in that domain (nurses, physicians, radiology . . . ) is likely to be productive.

**Tips for Discovering Stakeholders**

- Listen for names of people and organisations when you are starting on a project.
- Find out about the roles and organisations that people mention.
- Sketch an onion model in your notebook; add names and organisations to it; draw in relationships when you discover them.
- Use the template (Figure 2.4) to help identify missed roles.
- Once you have started analysing context, check your context knowledge against your stakeholder list.

## 2.4 Managing Your Stakeholders




**Answering the question:**
If there are so many people playing so many different roles in the project, how do we manage them all?

### 2.4.1 Engaging with Stakeholders

People are usually happy to have their views and opinions heard (see Chapters 11 and 12). The key to engaging with people on a project is simply to be open, honest and friendly, not to take sides, and to follow up each person's questions or suggestions appropriately. It also means taking great care with people's feelings. You need to keep track of your dealings with stakeholders, to analyse influences on your project, and to prioritise your activities accordingly. Managing stakeholders is more than discovering and documenting them – it's an ongoing negotiation.

**Table 2.2:** Basic stakeholder management list.

| Role | Name/ organisation | Contact details | Actions to be taken by project | Agreement status | Associated issues |
|---|---|---|---|---|---|
| Interfacing | Old Town Bus Company (OTBC) | Jane Brown (Business Planning) 151 Main Street Tel 0123–4567 | Negotiate position of bus/tram interchange near School Street | Preliminary; contact made | Effect of tram on bus passengers (increase or decrease) |
| Political | Anytown Council | Alderman Joe Bloggs (Transport Planning) The Town Hall 1 The Square Tel 0123–4121 | Agree outline approach | Council likely to give permission | Elections in September |
| ... | ... | ... | ... | ... | ... |

## 2.4.2 Keeping Track of Stakeholders

Stakeholders are the people who have influence on your project. That means you need to keep them happy. So, it's vital that you pay attention to them. At the very least, you need to know who they are, what they want, and whether they have agreed to the requirements. Table 2.2 illustrates a simple way of keeping track of your stakeholders.

Keep in mind that each stakeholder group (and individual stakeholders within groups) have different expectations. An important strategy is to keep stakeholders informed as the project proceeds. It is rarely possible to meet everyone's expectations. By providing periodic project status updates, you can manage people's expectations and improve the likelihood of project success.

## 2.4.3 Analysing Influences

Figure 2.5 illustrates a simple analysis of influences for a video game development project. Influences are drawn as an overlay of labelled arrows between stakeholders on an onion model.

This is an informal diagram, and the only rules are to draw whatever is useful for your project. (A mathematical analysis of influences is possible, but is unlikely to be either feasible or cost effective on industrial projects, as it requires extensive data collection.) In the diagram, a different colour or style of line is used for each kind of influence.

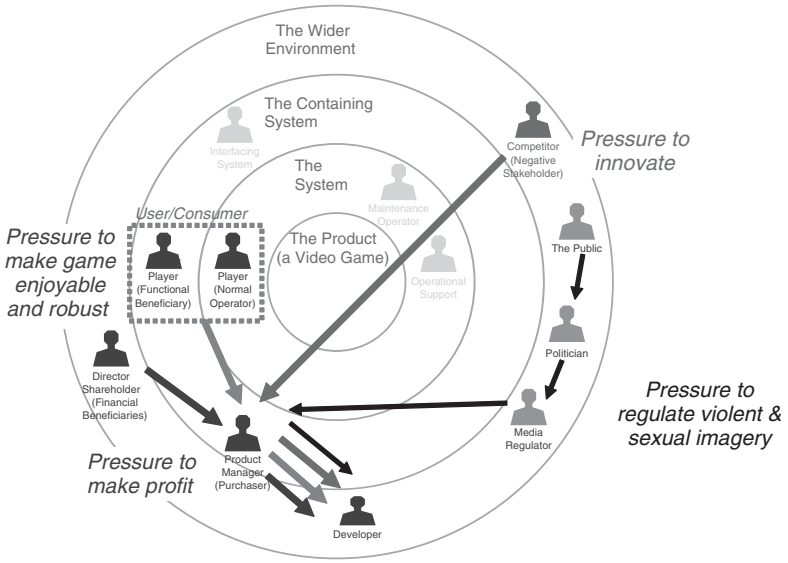



**Figure 2.5:** Analysing stakeholder influences on product development.

There are strong competing pressures on the maker of a video game:

- The consumer audience is savvy and aware of the market, and shares opinions freely on the Internet. In other words, reputations can be made or broken rapidly. Game players want new games to be exciting, scary, varied, emotionally intense, involving, realistic and action-packed. This all means development work and cost, on artistic design, music, game structure, interactions and so on.
- Competitors exert strong pressure to innovate or perish; the short period during which a game is new is its only opportunity to sell profitably.
- The public, through pressure on politicians, create media regulation of violent and offensive imagery.
- Meanwhile, the company's financial beneficiaries demand profit.
- The product manager, who is effectively the purchaser (the internal customer, representing the mass-market consumers), is responsible for integrating all these conflicting requirements into an effective specification, and transmitting these to the developers. This is requirements creation at the sharp end.

## 2.4.4 Prioritising Stakeholders

Projects often prioritise their stakeholder management activities by the relative power of the stakeholders. This is pragmatic, as long as you do not neglect less powerful stakeholders. One tried and tested approach is to make a matrix

of stakeholders against the kind of influence they have on the project: whether positive or negative, powerful or weak (politically), as in Table 2.3.

**A Consumer is Not a Typical Stakeholder**

Figure 2.5 illustrates the roles of 'user' or 'consumer'. These are not basic stakeholder roles like operator, regulator or functional beneficiary at all.

Instead, a player both operates a video game and benefits from it functionally (by enjoying the game). This hybrid role is called 'user'. Where the same person purchases the (mass-market) product, the role is a triple hybrid. Sometimes there is also a maintenance role involved.

Such all-in-one roles are not typical of stakeholder roles in general. Your personal experience of being a mass-market product consumer is not a good guide to understanding stakeholders in general.

As with any kind of risk register, the influence matrix is only of value if someone uses it to manage the project. On a project where influences matter, you should have a stakeholder manager who ensures that the requirements are understood and agreed, at least by the most powerful stakeholders. This role is closely related both to project management and to requirements management, so the task cannot be done in isolation.

**Table 2.3:** Influence matrix.

| | Negative | | Positive | |
|---|---|---|---|---|
| **Stakeholder** | **Strong** | **Weak** | **Weak** | **Strong** |
| Old Town Residents' Association | | At the moment, the OTRA is undecided. With careful integration, could become positive. | | |
| Anytown Council | | | | Currently very keen. Could change at local council elections in May. |
| Mansion Preservation Society | | Opposed to any route within 100m of Mansion; otherwise in favour. | | |
| ... | | | | |

### 2.4.5 Involving Stakeholders

You may need to choose your development approach to ensure that stakeholders are sufficiently involved in your project.

For example, you could choose a staged approach in which you carry out a feasibility study and obtain detailed feedback from stakeholders. You may then build a demonstrator, again obtaining feedback, before launching the main development process. The choice of life cycle depends on the nature of your project, the technological risk it entails and the stakeholder situation.

If you know of any negative or hostile stakeholder groups, you have an especially urgent task to make peace with them, or at least to defuse their negative feelings. Find out what they are concerned about; make sure they are properly informed, and set right any misconceptions they may have; and show that the project is professional and fair. If at all possible, take time to negotiate an agreed way forward with them.

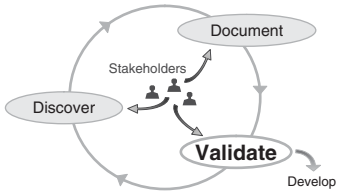
### 2.4.6 The Integrated Project Team

Where you are developing a product for use by a specific group of people – say, a healthcare team, engineers in the army or air traffic controllers – then the product is far more likely to be accepted if some representatives of that group form part of the development team.

An Integrated Project Team (IPT) includes people with all the skills and knowledge needed for the project to succeed. That may mean business analysts familiar with the domain, designers, programmers, human factors specialists, testers, and representatives of all the operational roles working with the product.

Creating an IPT sounds costly, but it need not be. IPT members do not have to be full time, as long as they are available often enough for communication to be easy and effective. Having operational stakeholders around to comment on a document or prototype as soon as it is made, and to suggest small changes that can be implemented in a few minutes, is far better (and, in the long run, cheaper) than relying on document deliveries and formal review comments.

## 2.5 Validating Your List of Stakeholders




Stakeholder groups are often fluid. People change their minds; companies are formed, merged or taken over; pressure groups become active and disappear.

For instance, in the context of the tram example, during an economic boom, property developers are very keen on new civic schemes such as shopping centres, and consequently (or in return for permission to build) they may be active supporters of transport schemes; they rapidly disappear, however, when times become harder.

Expect, therefore, to have to check and update your stakeholder knowledge regularly to keep your list 'complete':

- Make it a regular action on your project to check the stakeholder list.
- Regularly review actions on the stakeholder list and ensure they are taken.
- Update the stakeholder list to include people and organisations who have recently communicated with the project or who have been mentioned in project reports.
- Use the influence matrix to track changes to stakeholder positions.

### 2.5.1 Things To Check the Stakeholder Analysis Against

- Neighbours mentioned in the context model (Chapter 4).
- People mentioned in interviews (Chapter 11).

## 2.6 The Bare Minimum of Stakeholder Analysis

- Identify who could 'stop the show' on your project.
- Find out what those people want, and negotiate as necessary.

## 2.7 Next Steps: Requirements from Stakeholders

**Answering the questions:**

- OK, so we know who our stakeholders are: how does that help us?
- What do we do next?

Your stakeholders will lead you to many of your requirements, other than those you can discover for yourself by, for example, the analysis of existing systems or mathematical modelling.

How you'll create the requirements is the subject of the rest of this book. In general terms, Table 2.4 shows which stakeholders you will be most likely to work with to create each kind of requirement.

**Table 2.4:** Operational roles within 'the system'.

| Stakeholder role | Type of requirement | Discovery technique |
|---|---|---|
| Normal operators (possibly in many different roles) | Scenarios (Chapter 5)<br>Usability (Chapter 6) | Interview (Chapter 10)<br>Apprenticing (Chapter 10)<br>Observation (Chapter 10)<br>Workshops (Chapter 11)<br>Data modelling (Chapter 8)<br>Prototyping (Chapter 12)<br>Archaeology (Chapter 12) |
| Interfacing | Interface definitions (Chapter 4) | |
| Maintenance | Maintenance functions/scenarios (Chapter 5)<br>Diagnostics, built-in test | |
| Support | Support functions | |
| Functional beneficiary | Product functions/scenarios (Chapter 5)<br>Performance targets (Chapter 9) | |
| Financial beneficiary | Mission, objective (Chapter 3) | Interview (Chapter 10)<br>Read policy documents |
| Regulator | Regulations, laws, standards, guidance<br>Responses to safety case, compliance statements, etc | Legal advice on regulations, etc Negotiate compliance |
| Experts, specialists in disciplines | Safety, security, reliability, usability, etc (Chapter 6)<br>Constraints (Chapter 6) | Analysis, simulation, modelling, standards |
| Manufacturer | Producibility | Interview (Chapter 10)<br>Workshop (Chapter 11)<br>Prototyping (Chapter 12) |
| Marketing (surrogate, on behalf of mass-market customers) | Mass market (consumer)<br>Preferences by group (age, income, etc) | Market survey, Field trials<br>Observation (Chapter 10)<br>Prototyping (Chapter 12)<br>Analogous products (Chapter 12)<br>Competitor analysis |
| Product manager, purchaser | Priorities Programme, schedule Budget (cost) | Prioritisation (Chapter 13)<br>Trade-offs (Chapter 14) |
| The public | (Lack of negative impact) | Public meetings, Focus groups, Consultation, Roadshows |

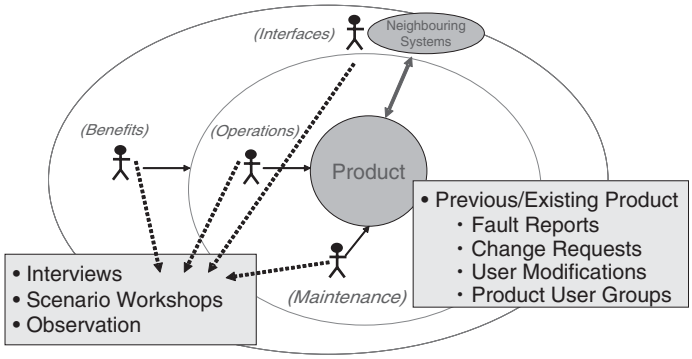



**Figure 2.6:** Requirements from operational stakeholders.

You will use a range of techniques described in Part I, such as goal and scenario modelling (see Chapters 3 and 5), to discover intended benefits and operational scenarios (Figure 2.6).You may also do some 'product archaeology' to discover possible requirements from fault reports and the like on the existing product.

You will similarly use a range of techniques and sources appropriate to your project to discover requirements with your non-operational stakeholders (Figure 2.7).

In each case, you should validate your findings with your stakeholders.

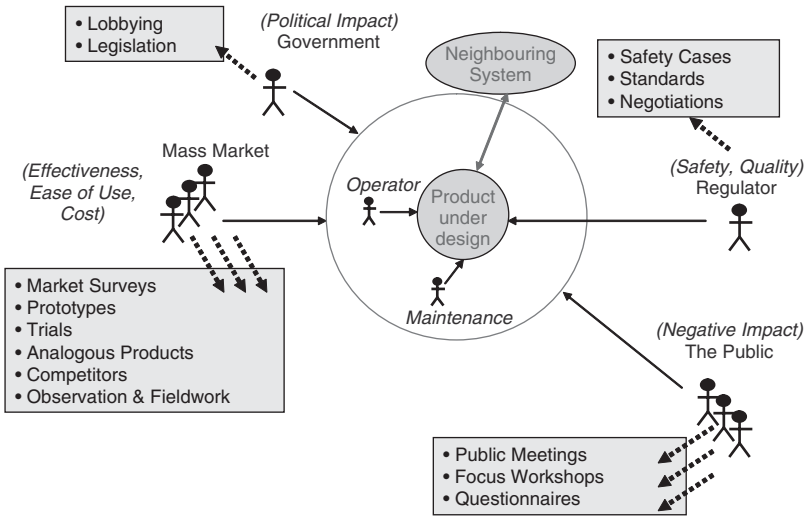



**Figure 2.7:** Requirements from non-operational stakeholders.

## 2.8 Exercise

You are a product manager for a machine tool company. The directors have asked you to develop a new cutting machine to cut cloth for fashionable dresses of all sizes and patterns. The machine will be sold to clothing makers around the world:

a. Who are your key stakeholders?
b. How will you analyse and validate your stakeholder list?

## 2.9 Further Reading

1. Alexander, I. (2005) A Taxonomy of Stakeholders: Human Roles in System Development, *International Journal of Technology and Human Interaction*, **1**(1), 23–59.
   Ian's paper on stakeholders is an academic analysis of stakeholder roles in a development project. It goes into more detail on each type of stakeholder than is possible here.
2. Robertson, S. and Robertson, J. (2004) *Requirements-Led Project Management: Discovering David's Slingshot*, Boston: Addison-Wesley.
   Chapter 3, on project sociology, in the Robertsons' excellent book provides several different points of view on project stakeholders, including Belbin's team roles, Ian's onion model, and looking for the knowledge on different areas, such as security.