Software requirements

Styles and techniques

Soren Lauesen

Addison-Wesley

An imprint of Pearson Education

London/Boston/Indianapolis/New York/Mexico City/Toronto/Sydney/Tokyo/Singapore
Hong Kong/Cape Town/New Delhi/Madrid/Paris/Amsterdam/Munich/Milan/Stockholm

Contents

1.2 Project types

Highlights

There are many types of project: in-house development, buying commercial software, tenders and contracts, etc.
Requirements have different roles in different types of projects.
Sometimes the project type is unknown.

Requirements play somewhat different roles in different types of projects. Here are some typical projects (also see Figure 1.2):

In-house development. The system is developed inside a company for the company's own use. The customer and the supplier are departments of the same company.

This situation often occurs in larger companies, for instance in banks, insurance companies, and large manufacturing companies. Traditionally, these projects are carried out without specified requirements, and many projects end in disaster. To avoid this, many companies now try to use written requirements.

Product development. The product is a commercial product to be marketed by a company. The development project is carried out inside the company. The "customer" is the marketing department and the "supplier" the development department.

The parties often use requirements on many levels, for instance market-oriented requirements stating the demands in the market, and design-level requirements that specify exact details of the user interface. A frequent problem in these situations is that developers never see the real customers (those buying the product). Developers have to rely on the translation of requirements by marketing. We have seen many cases where this has caused tremendous problems in development as well as in market acceptance of the product.

Time-and-material based development. A software house develops the system on a time-and-materials base, i.e. the customer pays the costs, for instance month by month. Requirements are informal (unwritten) and develop over time.

In the old days this was the usual way to develop systems, and it still thrives in some areas, for instance some Internet software houses. Over time, the parties tend to realize that such projects can easily get out of control. The costs skyrocket, and the parties fight over what has to be delivered and who is to blame.

Introducing written requirements is an improvement, since it becomes clear what is to be delivered. The supplier may still develop the system on a time-and-materials basis, so the costs are only partly controlled. In this version, we could just as well call the project a contract-based development with a variable price.

Fig 1.2 Project types

Project types	Customer	Supplier
In-house Product development Time and materials	User dept. Marketing Company	IT dept. Software dept. Software house
COTS	Company	(Vendor)
Tender Contract development Sub-contracting	Company Company Supplier	Supplier Software house Software house
Unknown		Inhouse? COTS?

COTS purchase. COTS is an acronym for Commercial Off The Shelf and denotes a commercial package that you can buy – more or less off the shelf. Examples are Microsoft Office, development tools, Lotus Notes, large business applications (e.g. SAP, BAAN, or Navision), commercial bank or hospital systems, and so on.

Some of these products are fully off the shelf (e.g. Microsoft Office and some development tools). Others can be configured in so many ways that the customer needs a consultant to do it (e.g. SAP). Many COTS systems can be extended according to the special needs of the customer (e.g. bank or hospital systems), and some COTS systems are only frameworks where all the customer logic has to be built in (e.g. Lotus Notes).

The large business applications (SAP, BAAN, and so on) are often called ERP systems, Enterprise Resource Planning systems.

When we use the term *COTS purchase* in this book, we mean the purchase of a fully off-the-shelf product. Whatever extensions and configurations are required, the customer takes care of them himself. What role do requirements have in this case? Here, you don't set up a formal contract with requirements no more than when you buy a fridge, but it is still a good idea to figure out what your needs are so that you – in a structured way – can choose between the alternatives. Some forms of requirements are suited to that, as we will see throughout the book.

When we want a COTS product including some tailor-made configuration or extension, we use the term *COTS-based acquisition*. Requirements and contracts are very important here, so that you get the additional things that you need. The product you get consists of the COTS-part itself plus the tailor-made things. The situation can be handled as a tender project or a contract development. For a discussion of various types of COTS projects, see Carney and Long (2000). For a discussion of changed development processes to cater for COTS, see Brownsword *et al.* (2000) and Maiden and Ncube (1998).

Tender. The customer company starts a tender process and sends out a request for proposal (RFP). Several suppliers are invited to submit proposals. The tender documentation contains an elaborate requirements specification. The customer writes the specification or asks a consultant to do it. Later the customer selects the best proposal.

This approach is used more and more often today, particularly by government organizations, since they have an obligation to treat all suppliers equally and avoid any hint of nepotism or corruption.

Writing requirements for these situations is a real challenge. You have to write for a broad audience that you don't know, and you end up signing a contract for delivery, based primarily on the supplier's reply to the requirements. In many countries the customer is not allowed to bargain over features and prices after having sent out the request for proposal.

The tender approach is used for tailor-made systems as well as COTS-based systems. For the COTS parts of the system, you cannot specify product details such as screen pictures, since the screen pictures are largely determined by the product, which you of course have to choose later. This makes COTS tenders even more difficult from a requirements viewpoint. See sections 7.3 to 7.5 for more about tender projects.

Contract development. A supplier company develops or delivers a system to the customer company. The requirements specification and the contract specify what is to be delivered. The two parties will often work together for some time to write the requirements and the contract.

The system may be tailor-made or a COTS-based system with extensions.

This approach has been used for a long time in public as well as in private organizations. It is often a good idea to make two-step contracts where the first contract deals with making the detailed requirements, while the next one deals with developing the system. Prices can be fixed or variable, or a combination. See more on this in section 1.7.4.

Sub-contracting. A sub-contractor develops or delivers part of a system to a main contractor, who delivers the total system to a customer. Sub-contracting can be requirements-based or time-and-materials based without written requirements.

Sub-contracting differs from other situations in that both parties usually speak the "IT-language" and agree on technical interfaces between their products. Often the main contractor has several sub-contractors who develop different parts of the total product. Sections 5.4 and 5.5 discuss these situations.

Situation unknown. In many cases the customer doesn't know what he should do. Should he buy a COTS system or have the system developed in-house? Should he try to play the role of a main contractor and integrate several products, or should he contract with someone else to do it? See Sikkel *et al.* (2000) for a discussion of such situations.

High-level requirements can help to resolve these issues. They may help compare the alternatives from a cost/benefit and risk perspective, and they may help in looking for potential suppliers in-house or outside. The membership system in Chapter 15 handles such a situation.

1.6 The goal–design scale

Highlights

Goal-level requirement: why the customer wants to spend money on the product.
Domain-level requirement: support user tasks xx to yy.
Product-level requirement: a function to be provided by the product.
Design-level requirement: details of the product interface.

Tradition says that a requirement must specify

what the system should do

without specifying how.

The reason is that if you specify "how", you have entered the design phase and may have excluded possibilities that are better than those you thought of initially. In practice it is difficult to distinguish "what" from "how". The right choice depends on the individual situation.

We will illustrate the issue with an example from the Danish Shipyard (Chapter 11). The shipyard specializes in ship repairs. The values of orders range from $10,000 to $5 million, and as many as 300 workers may be involved in one order. Competition is extremely fierce and repair orders are usually negotiated and signed while the ship is at sea.

The management of the shipyard decided, for several reasons, to replace their old business application with a more modern one. One of their business goals was to achieve a better way of calculating costs.

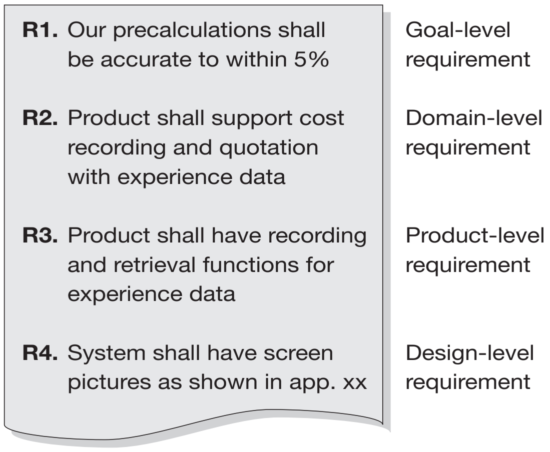
When preparing a quote, the sales staff precalculate the costs, but often the actual costs exceed the precalculation, causing the shipyard to lose money. Or the precalculated cost is unnecessarily high, causing the shipyard to lose the order. What is the solution to this? Maybe the new IT system could collect data from earlier orders and use it to support new cost calculations. Experience data could for instance include the average time it takes to weld a ton of iron, the average time it takes to paint 100 square meters of ship, etc.

Figure 1.6A shows four possibilities for the requirements in this case, which we will discuss one by one.

Goal-level requirement

R1 The product shall ensure that precalculations match actual costs within a standard deviation of 5%.

Fig 1.6A The goal-design scale



R1. Our precalculations shall be accurate to within 5%	Goal-level requirement
R2. Product shall support cost recording and quotation with experience data	Domain-level requirement
R3. Product shall have recording and retrieval functions for experience data	Product-level requirement
R4. System shall have screen pictures as shown in app. xx	Design-level requirement

Which requirement should be chosen if the supplier is:

- A vendor of business applications?
- A software house concentrating on programming?
- PriceWaterhouseCoopers?

This requirement states the business goal, which is good because that is what the shipyard really want. Note that we call it a goal-level requirement because it is a business goal that can be verified, although only after some period of operation. Unfortunately, if you ask a software house to accept this requirement, they will refuse. They cannot take the responsibility for R1, because it requires much more than a new IT product: it is also necessary to train and motivate the shipyard staff, build up an experience database, etc., and even then it may be impossible to reach the goal. The customer has to take responsibility for that.

Domain-level requirement

R2 The product shall support the cost registration task including recording of experience data. It shall also support the quotation task with experience data.

This is a typical domain-level requirement. It outlines the tasks involved and requires support for these tasks. The analyst has carefully identified the right tasks. For instance, he hasn't specified a new user task to record experience data, because his knowledge of the shipyard and its day-to-day work tells him that then the recording would never be done. It must be done as part of something that is done already – recording the costs. Sections 3.6 and 3.8 explain more about domain-level requirements.

Could we give this requirement to a software house? That depends. If it is a software house that knows about shipyards or similar types of businesses, it may work. It doesn't matter whether the software house offers a COTS-based system with the necessary extensions, or whether they develop a system from scratch. However, if we choose a software house that is good at programming, but doesn't know about business applications, it would be highly risky, because they may come up with completely inadequate solutions.

Can we verify the requirement? Yes, even before the delivery time. We can try to carry out the tasks and see whether the system supports it. Deciding whether the support is adequate is a matter of assessing the quality. We discuss this in section 7.3.

What about validation? Can the customer reach his business goals? We can see that there is a requirement intended to support the goal, but we cannot be sure that it is sufficient. Here the customer runs a risk, but that is the kind of risk he should handle and be responsible for: he cannot transfer it to the software house.

Product-level requirement

R3 The product shall have a function for recording experience data and associated keywords. It shall have a function for retrieving the data based on keywords.

This is a typical product-level requirement, where we specify what comes in and goes out of the product. Essentially we just identify the function or feature without giving all the details. Section 3.4 tells more about this kind of requirement.

Could we give the requirement to a software house? Yes. If it is a software house that knows about shipyards there is no problem. Using COTS or developing from scratch are both acceptable. If we choose a software house that doesn't know about business applications, we would have to add some more detail about experience data, keywords, etc., then they should be able to provide the features we have asked for. Can we verify the requirement? Yes, before the delivery time. All that needs to be done is for us to check that the necessary screens are there and that they work.

What about validation? Here the customer runs the same risk as for R2. However, we run an additional risk. We cannot be sure that the solution adequately supports the tasks. Maybe the supplier has designed the solution in such a way that the user has to leave the cost registration screen, enter various codes once more, and then enter the experience data. A likely result would be that experience data isn't recorded.

Design-level requirement

R4 The product shall provide the screen pictures shown in app. xx. The menu points shall work as specified in yy.

This is a typical design-level requirement, where we specify one of the product interfaces in detail. Although a design-level requirement specifies the interface exactly, it doesn't show how to implement it inside the product.

R4 refers to the shipyard's own solution in app. xx. If they asked a business system supplier for R4, they might not get the best system. A supplier may have better solutions for experience data, but they are likely to use different screen pictures than those in app. xx. Insisting on the customer's own screen pictures might also be much more costly than using an off-the-shelf solution.

However, if the product was a rare type of system, the shipyard might have to use a software house without domain knowledge and have them develop the solution from scratch. In that case, R4 might be a very good requirement, assuming that the shipyard has designed the solution carefully. The shipyard would thus have full responsibility for ease of use, efficient task support, and its own business goals.

Choosing the right level

The conclusion of the analysis is: *choosing the right level on the goal-design scale is a matter of who you ask to do the job.*

You should not give the supplier more responsibility than he can handle. He may refuse to accept the added responsibility, or he may accept it but deliver an inadequate solution. Neither should you give him too few choices. It may make the solution too expensive, and if you haven't validated the requirements carefully, you may get an inferior solution.

In practice, the shipyard case is best handled through R2, the domain-level requirement. The main reason is that R2 ensures adequate task support and allows us to choose between many COTS suppliers. However, R1 is still important, although not as a requirement, but as a measurable goal stated in the introductory part of the spec. R4 may also be a good idea, not as a requirement, but as an example of what the customer has in mind. Of course, the customer shouldn't spend too much work on R4 since it is only an example.

R3 is rarely a good idea. The customer runs an unnecessary risk of inefficient task support and missed goals. Unfortunately, most requirements specs work on that level, and it is often a source of problems.

In the discussion above, we discarded R1, the goal-level requirement, because a software house couldn't take responsibility for it. Could we find a supplier that could accept this requirement? Maybe, but we would have to use a completely different type of supplier, for instance a management consultant such as PriceWaterhouseCoopers, Ernst & Young, etc. In their contract with the consultant, R1 would be the requirement, and R2 would be an example of a possible (partial) solution.

It is, however, likely that not even the consultant would accept R1 at a fixed price. Instead he might work on a time-and-material basis, tell the customer about other solutions and advise him whether experience has shown that 5% deviation was achievable in a shipyard, how to train staff, etc. In essence, the customer would get an organizational solution, possibly including some IT.

Quite often it is a good idea to use different requirement levels for different interfaces, or change from one level to another during the project. In section 1.7 and Chapter 5 we show ways of combining the levels.

What happened in the Danish Shipyard?

You may wonder what actually happened in the Danish Shipyard case discussed above. The full requirements specification is in Chapter 11. There are eight business goals, clearly stated in section 4 of the spec, which is an introductory section intended to help the reader. The real requirements are in section 5, which uses product-level requirements similar to R3 above. (Each sub-section has also an introduction that explains a bit about the tasks involved.)

When the system had been in operation for about a year, we assessed it and noticed that sales staff didn't use the experience data. Why? The financial manager, who had been closely involved in the development, said it was because sales staff were reluctant to use computers and they should be replaced by younger staff. A closer study revealed, however, that the system had no features to enable it to record experience data, and the feature for retrieving experience data was difficult to find and use from the quotation windows.

Further study revealed that the requirements didn't mention recording of experience data – it had accidentally been replaced by a reference to *retrieval* of experience data. The spec mentioned retrieval of experience data but didn't make it clear that this should be done during quotation. So, although all the requirements were verified at delivery time, nobody noticed that a highly important goal was lost.

How could this have been avoided? One possibility would be to trace the eight business goals to requirements and check them several times during development to ensure that the project remains on-track. A good quality manager would have ensured this. Another possibility would be to use domain-level requirements rather than product-level ones.

Asking "why"

In practice it is not always easy to choose the right level on the goal-design scale, and analysts often make mistakes. They should try to slide up and down the scale by asking "why" each requirement is necessary and "how" it can be achieved. Then they can select the requirement appropriate for the project.

Figure 1.6B shows some "why" questions for a device that measures neural signals in human patients. The customer developed medical equipment and had sub-contracted the software and part of the hardware development. One requirement said that the product should have a special mini-keyboard with start/stop button, repeat button, and a few other buttons.

Fig 1.6B Ask “why”

Neural diagnostics

System shall have mini-keyboard with start/stop button, ...
Why?

So that it is possible to operate it with the “left hand”
Why?

Because both hands must be at the patient
Why?

To control electrodes and bandages, and to calm and reassure the patient.

The supplier could deliver that, but just to make sure he understood the requirements, he asked “why” the mini-keyboard was necessary. The answer was that it should be possible to operate the device with “the left hand.” Still a bit puzzled, the supplier asked “why” once more.

The customer explained that, “both the surgeon’s hands have to be at the patient, so the surgeon couldn’t use an ordinary keyboard.”

“Why?” insisted the supplier.

“Because the surgeon has to fix electrodes to the patient’s body, and sometimes keep bandages in place, and at the same time calm the patient because the electrical signals can be painful,” was the answer.

Now the supplier understood the goal, but also understood that the requirements were too design-oriented. Since the supplier also supplied part of the hardware, more freedom was possible. The most important point was that both the surgeon’s hands had to be at the patient.

The parties could now see various other ways to satisfy the real need, for example with a mini-keyboard attached to the surgeon’s wrist, with foot pedals, or with voice recognition and a headset microphone.

They ended up with the requirement outlined in Figure 1.6C. Note that it includes both a domain description and some examples in order to help the reader and to record the ideas generated at an early stage.

4.1 Complex and simple functions

Highlights

Ignore trivial functions.
Describe semi-complex domain functions in requirements.
Describe semi-complex product functions as part of design.
Specify complex functions by performance or standards.

When do you need to describe a function in detail? You might believe that it is when a function is complex, but that is not the correct answer. In general it only makes sense to describe the semi-complex functions. It is waste of time to describe the simple functions, and the really complex functions must be handled in indirect ways.

Figure 4.1 summarizes these situations. Essentially there are two problem dimensions, one dealing with the obviousness of the domain and one dealing with the complexity of the program or algorithm.

Whether some part of the domain is obvious to the developer or not is not a clear-cut distinction. The analyst has to know the kind of developers involved, and if in doubt, he should assume that the domain part is non-obvious. Do we need the distinction?– Couldn't we just assume that it is all non-obvious? In principle yes, but in practice we have to distinguish in order to save time and in order to avoid hiding all the important things in trivialities. Remember: We have to live with tacit requirements.

Domain obvious, simple program

If the domain is obvious, it is clear to the programmer what the function should do. If the corresponding program part is simple, programming the function is straightforward. Fortunately, most functions are of this kind. Examples are FindFreeRoom, RecordBooking, PrintInvoice. No details of the function are needed. It is sufficient to find a good title for the function, provide some understanding of the context in which it will be used, and sometimes specify the input and output formats.

Domain obvious, interaction complexity

If the domain is obvious, it is also obvious to the programmer what the function should do. However, the interaction between several functions may make the program somewhat complex. An example is the Check-in function, as it must work differently depending on the situation, i.e. depending on other functions that have been used before Check-in.

Fig 4.1 Complex and simple functions

	Domain obvious	**Domain non-obvious**
Simple program	FindFreeRoom PrintInvoice	Discount calculation Business rules

Interaction complexity	Checkin
	if booked
	if non-booked
	if add room
		Tax calculation
		Payroll

Hard to program	Fastest truck route Voice recognition?	Optimize roster Voice recognition?

Possible requirement styles:
A. (Leave to intuition)
B. Natural language or tables
C. Process description (algorithm)
D. Performance specification
E. Refer to laws and agreements

Suitable choices?

Since the domain aspect is simple, this is not really a requirement issue, but a design and programming issue. However, if we design the user interface as part of the requirements, we have to deal with interaction complexity in the user interface functions. The rest of this chapter shows many ways of specifying this complexity, ranging from state diagrams and textual process descriptions to activity diagrams and sequence diagrams.

Domain obvious, hard to program

In some cases the domain is obvious, yet the function is hard to program. Here are two examples:

Fastest truck route. Given a database of road sections and travel times per section. Find the fastest truck route between point X in city A and point Y in city B.

Everybody can understand what this function should produce as output. It is somewhat difficult to program the function for a small road network, but a good programmer should be able to handle it. However, if we talk about large national networks with a million roads, it is very hard to obtain a reasonable response time. Only a few specialists in the world are able to handle this.

If we specify requirements for a system that includes such functions, we should not attempt to specify the algorithm. Instead we should specify the performance of the system, e.g. the response time and the size of the network. These requirements are quality requirements, however, and not functional requirements.

If we know an expert in the area or know a paper that describes a solution, we might help the supplier with a reference to these sources. This is helping the reader, of course, and not a requirement.

> **Voice recognition.** Given the digital recording of a spoken sentence, print the sentence. Or a simpler variant of the above: recognize the statement as a request to push some button, and then push that button.

Everybody knows what the result of such a function should be. At present, however, nobody can fully implement the function. Some software suppliers have programs that can do this, but with limitations: only for a particular language, a particular group of speakers, when spoken carefully, and with little background noise. Even under these conditions, the program has a limited hit-rate.

If we want to develop a system that includes such features, we shouldn't attempt to specify the algorithm. We should instead specify the precision of the system, for instance the expected hit-rate for a certain group of speakers with a certain level of background noise. Again these requirements are quality requirements rather than functional requirements.

Domain non-obvious, simple program

When the programmer has inadequate knowledge of the domain, he cannot program the function on his own. However, once he understands the domain, the function is fairly simple to program. Here are some examples:

> **Discount calculation.** Business customers usually get discounts depending on the kind of product they buy, the order size, and the total yearly size of their purchases.
>
> **Business rules.** Companies usually have many more or less explicit rules, for instance about when a deposit is needed, when the supervisor's signature is needed, etc.

Sometimes there are written rules that cover these functions. It might then be sufficient to refer to those rules, but as an analyst you should check that the rules are also adequate from a developer's perspective. A good way is to work through a couple of examples with an expert user in order to see that the written rules are unambiguous. Usually, you will have to write some additional comments or write down some examples to guide the developer. These examples also make good test cases later.

In other cases, the rules are shared oral knowledge. Sometimes you don't have to care about the rules if the user interface allows the user to exercise his usual judgment. At other times you have to write them down and check them with expert users. One solution is to make a table of the possibilities and the outcomes (section 4.2 has an example), or you may write a textual process description (section 4.3), or you may draw activity diagrams (section 4.6), state diagrams, etc.

Domain non-obvious, interaction complexity

In some cases a typical programmer may have inadequate knowledge of the domain, and it would take him a long time to get information about all the rules that apply. Although each rule is simple, it is not straightforward to make the program because the many rules interfere with each other. Here are some examples:

Tax calculation. Given a person's or a company's income, expenses, etc. compute the tax to be paid.

Payroll calculation. Given the necessary information about work hours, age, job type, etc. calculate the pay.

These functions are not computationally complex, but it is very difficult to find out exactly what they should do because there are so many rules to handle. You have to study tax laws or union regulations and extract the necessary algorithms. As an example, in some countries union regulations for wages are extremely complex, and not even the workers themselves are sure what their salary should be.

Should the analyst extract this information and specify the algorithm? Usually not; unless he is a specialist in that area he cannot do it properly.

Suppliers that develop tax programs or payroll programs usually have an entire department for the purpose. These people have law and domain expertise, and they spend most of their time translating laws and agreements into programs. A requirements specialist requiring tax calculation as a part of a larger system should not attempt to compete with these specialists, but should use them as suppliers or sub-contractors.

Usually the solution is to write a requirement referring to laws or standards (see section 3.15).

Domain non-obvious, hard to program

In some cases a typical programmer has inadequate knowledge of the domain and, even if he knew the domain, the algorithmic complexity would be high. These situations are frequent and are important for both business and society in general. Here is an example:

Roster optimization. Calculate rosters for hospital staff so that adequate capacity and qualifications are ensured at all times and total wage expenses are minimized.

Sounds easy? It is not. First of all, you need a lot of domain knowledge about hospitals. You also need knowledge of all the union regulations that influence the wage. Given all this knowledge, the problem has also a very high computational complexity. In principle, you could let the computer generate all possible rosters, check whether it fulfills the rules, calculate the wage expense for that roster, and then simply select the best roster. Unfortunately, there will typically be more than a trillion possible rosters, so finding the best among them in this simple way might take a fast computer more than a thousand years.

There are many ways to deal with the computational complexity. Tricky optimization algorithms can sometimes cut the run time by a factor of millions. Being satisfied with sub-optimal solutions may also help a lot. Making an interactive system where computer and user co-operate to produce a good roster can be significantly better than the present manual system. But all of this needs a concerted effort between several domain experts and algorithm specialists. So what should be written in the specification? You may simply write a requirement like this:

R1: The system shall calculate rosters for hospital staff so that adequate capacity and qualifications are ensured at all times and total wage expenses are kept low. The solution might work as a close interaction between user and system. The supplier should specify how much better the rosters are than typical manual rosters.

This requirement assumes that the supplier has the necessary domain and algorithm expertise. If there are several potential suppliers, select the best one based on the quality of their solution. It is possible to verify that the quality is as promised. In case of COTS-based solutions, it can even be done before signing the contract.

Voice recognition. In Figure 4.1, we have put the voice recognition problem in both the obvious and the non-obvious domain categories. Its position depends on how much domain expertise the developer needs. There are simple approaches for identifying single words, where the word is compared against a large collection of spoken words to find the best match. These solutions do not require domain knowledge, only the algorithmic complexity must be mastered. On the other hand, if we want sentences to be recognized, a lot of linguistic and phonetic knowledge is needed, so a combination of domain expertise and algorithmic expertise is needed.

From a requirements viewpoint, it is simply a matter of specifying the required accuracy level for the voice recognition. The problem comes when the customer approaches potential suppliers. Will he need a specialist in algorithms, or a specialist in linguistics, or both?

6.3 Open metric and open target

Highlights

Often hard to select a metric for measuring quality.
Even harder to decide the value needed.
Solution: Leave it to the supplier to specify it.

A quality requirement will often have a numerical target to reach. Figure 6.3A shows a simple example from a speed trap system where speed violators are photographed for identification:

R1 The product shall detect a speed violation and take a photo within 0.5 seconds.

The target of 0.5 seconds has a physical explanation. If the trap system doesn't take the photo within that time, the violator will have disappeared from the camera view. Furthermore, a faster system has no extra advantage.

In many cases, however, there is not a similar justification for the target value. Consider the next example in Figure 6.3A:

R2 The product shall compute and show a forecast for room occupation within two minutes.

This requirement specifies the target "two minutes" for the response time. It is easy to specify such a target, but from where do we get it? What would happen if we wrote four minutes? And why not ask for one minute?

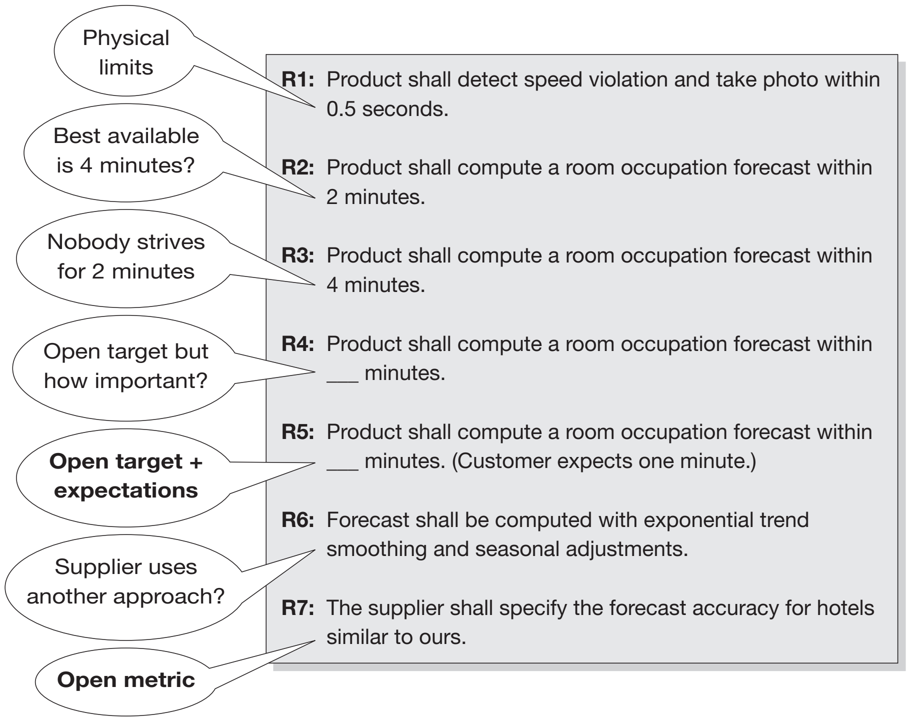
This is a typical case where the quality requirement is not really mandatory. Assume that nobody could guarantee a response time of two minutes (at a reasonable cost), but four minutes were available. In this case, the customer would probably accept the four minutes.

The customer might even choose a system with a four-minute response time, even though one with two minutes was available. This could happen if the four-minute system was superior in other respects.

Apparently, the customer can live with the higher limit, so why did he specify a low one? All right, he should specify four minutes, as in R3 on the figure. Unfortunately, this means that the supplier will not try to make an effort to get down to two minutes, so the customer gets a worse system. Furthermore, if the customer specified four minutes, by the same argument he might still accept a higher limit if necessary.

Open target One way out is to ask the supplier to specify what response time he can provide, as shown in R4. We call this the *open target* approach.

Fig 6.3A Open metric and open target


Physical limits
Best available is 4 minutes?
Nobody strives for 2 minutes
Open target but how important?
Open target + expectations
Supplier uses another approach?
Open metric

R1: Product shall detect speed violation and take photo within 0.5 seconds.

R2: Product shall compute a room occupation forecast within 2 minutes.

R3: Product shall compute a room occupation forecast within 4 minutes.

R4: Product shall compute a room occupation forecast within ___ minutes.

R5: Product shall compute a room occupation forecast within ___ minutes. (Customer expects one minute.)

R6: Forecast shall be computed with exponential trend smoothing and seasonal adjustments.

R7: The supplier shall specify the forecast accuracy for hotels similar to ours.

There is a problem with that. The supplier has no idea how important the response time is. Some suppliers may suggest a very expensive solution to provide a short response time, others may suggest a cheap solution with a response time they consider adequate.

How do we handle that? We see good results with the approach shown in R5. The customer specifies his expectations and asks the supplier to specify what he can offer. In a contract situation, this will usually start a discussion of what is feasible. In a tender situation, the customer may compare several proposals and select the “best” according to an overall evaluation. Skilled suppliers may even include options with separate prices in their proposal, for instance an option with a fast response time and one with a slow response time.

Open metric In some cases the customer may not even know how to measure the quality of the system. He hasn’t defined the metric to be used. R6 and R7 on Figure 6.3A shows an example of this.

6.5 Performance requirements

Highlights

Many kinds: response time, peak traffic, etc.
Technical and psychological limits.
Risky to mix up average cases, 95% cases, and worst cases.

Performance requirements specify how fast the product shall be, for instance the response time for various functions. The requirements are product-related and technical.

We can also specify domain-related performance requirements, e.g. the maximum time it must take an experienced user to perform a specific task. This is a highly relevant requirement. The total task time consists of product response times and human actions. By convention, such task-related requirements are considered usability requirements (see section 6.7, task time).

Figure 6.5A shows examples of performance requirements specified in various ways.

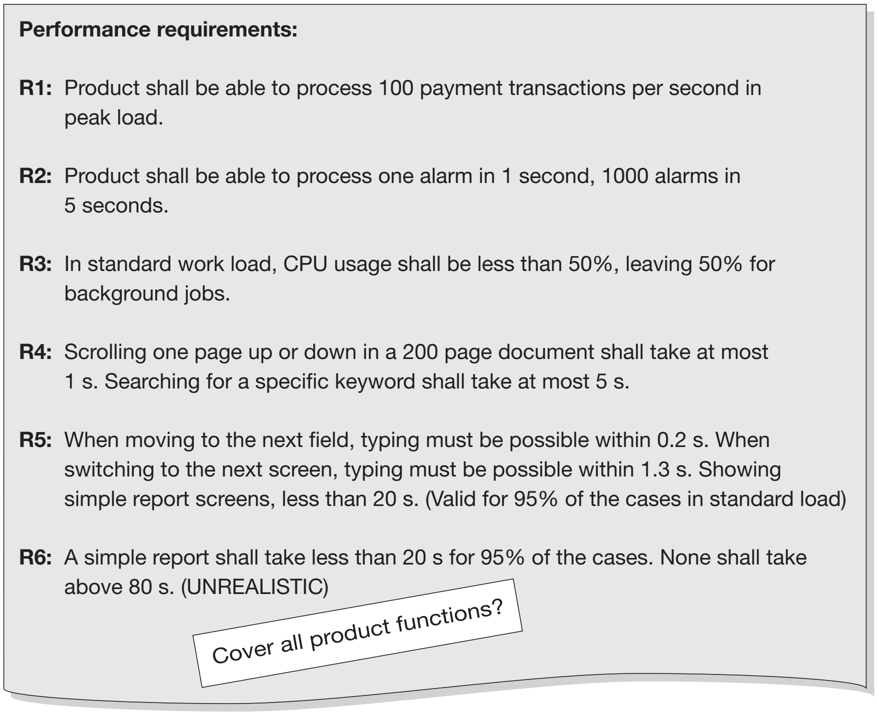
R1, payment transactions. R1 is from a system that handles credit card transactions. A transaction consists of a few messages. R1 specifies the *throughput*, that is the number of transactions to be handled per unit of time. The specification doesn't say anything about response time, and temporary delays are completely acceptable as long as the average over some longer period is within the specified limit. The term *peak load* should be explained somewhere else. It could be a special setup where no other activities are going on or a particularly busy day of the year.

R2, remote process control. R2 is from a system controlling electrical power in a geographical area. An alarm is a message telling about a power malfunction in the area. The user has to decide what to do when the alarm arrives. In a specific case, the customer expected about 1500 alarms a year, so one second per alarm seemed sufficient.

However, if asked about the real critical tasks, the customer would have explained that those were when something serious happened, like a mast falling in a winter storm. In those cases one malfunction would trigger a few hundred alarms within milliseconds. If R2 didn't mention the 5 seconds for 1000 alarms, the customer might get a system where the operator would have to wait one second for each of the several hundred alarms before he could do anything.

In this case, I was one of the developers involved. We had very little domain knowledge and forgot to ask about the critical tasks. As a result, R2 was not properly stated. Our system handled alarms very reliably, but only one per second. One winter night, a high-voltage mast fell, and about 600 alarms were sent. The operator couldn't see what to do until he had a survey picture reflecting all 600

Fig 6.5A Performance requirements



Performance requirements:

R1: Product shall be able to process 100 payment transactions per second in peak load.

R2: Product shall be able to process one alarm in 1 second, 1000 alarms in 5 seconds.

R3: In standard work load, CPU usage shall be less than 50%, leaving 50% for background jobs.

R4: Scrolling one page up or down in a 200 page document shall take at most 1 s. Searching for a specific keyword shall take at most 5 s.

R5: When moving to the next field, typing must be possible within 0.2 s. When switching to the next screen, typing must be possible within 1.3 s. Showing simple report screens, less than 20 s. (Valid for 95% of the cases in standard load)

R6: A simple report shall take less than 20 s for 95% of the cases. None shall take above 80 s. (UNREALISTIC)

Cover all product functions?

alarms, and that took 10 minutes. Then it was easy. He just had to reroute power through other high-voltage lines and everybody would have power again. That night, however, 200,000 people had no power for 12 minutes. The reason for this? We had no domain knowledge and forgot to ask the right questions. Repairing the problem after delivering the system was very difficult because our program logic was geared to the first-come first-served logic.

R3, multi-tasking environment. R3 is from a multi-tasking system that can perform several activities concurrently. The product must not monopolize the CPU. 50% of CPU time should be available for other activities. (Some analysts would for good reasons insist that this is a capacity requirement, because it defines a limit for how much CPU capacity the product may consume.)

R4, shipyard invoices. R4 is from a shipyard. An invoice after a ship repair may be more than 100 pages and it looks more like a novel than a list of items. Editing the invoice is a major task. The requirement reflects the bad experience that the users had with their present invoicing system.

R5, multi-user business system. R5 is from a typical business application. It distinguishes three kinds of functions and defines a maximum response time for each kind. (A more precise specification of the three kinds of functions might be necessary in some types of projects.)

Psychological limits

The three limits, 0.2, 1.3, and 20 seconds, are not chosen at random, but are based on human psychology. An average typist types around 5 characters a second, so 0.2 seconds for moving to the next field allows continuous typing. When experienced users switch to the next chunk of typing, they mentally prepare themselves for 1.35 seconds, for instance to read the next chunk of data. A screen switch of 1.3 seconds will thus not delay users. Read more about this keystroke-level model in section 10.6.

Finally, if the user has to wait more than 20 seconds, he will look for something else to do – even if the system warns him of the likely delay. The result is a significant waste of time because task changes often require a large mental effort (Solingen *et al.* 1998).

R5 solves the problem of specifying response times for a large number of screens – screens that we probably aren't aware of yet. We have also common justifications for the limits in R5. However, deviations from the general limits may be acceptable for some functions that need more time for technical reasons. In such cases, simply state the response time for the functions in question. To avoid apparent inconsistency, add to R5 that the general limits apply where nothing else is stated.

Average and upper limits

Notice that R5 doesn't insist on the stated response time in all cases – 95% is enough. Why is this necessary? Couldn't we just raise the response time a bit and say that it should be valid in 100% of cases? Quite often we see such requirements. R6 is a typical example:

R6 A simple report shall take less than 20 seconds in 95% of cases. None shall take above 80 seconds.

Usually such a requirement would be extremely expensive. Let us see why.

In a multi-user system, response times will vary due to coincidence between user actions. Assume for instance that the system can handle a request in 8 seconds if it has nothing else to do. Also assume that 100 users may use the system and send requests to it every now and then.

If all users by coincidence send a request at almost the same time, their request will go through the same pipeline. The first user will get a response within 8 seconds and the last a response within 800 seconds. Wow, we wanted 80 seconds in all cases! If we really insist on that, we would need hardware that is 10 times as fast to cut the time down from 800 seconds to 80. Such a system would be awfully expensive.

Response time probabilities

However, the worst case is extremely rare. Maybe the customer would be satisfied if we only exceed the 80 seconds very rarely. How often would it happen? We cannot answer this without knowing how many requests the system has to serve. If users send requests very often, a queue of requests will build up and response times increase. If requests are rare, the queues will be short and we will only exceed the 80 seconds very rarely.

Figure 6.5B shows how we can answer the question. In the example we assume that a request arrives to the system every 10 seconds on average – the inter-arrival time is 10 seconds. If all 100 users are equally busy, each of them will thus send a request every 1000 seconds on average (every 17 minutes). We now have this basic data:

Service time:	8 seconds
Inter-arrival time:	10 seconds
System load:	8/10, i.e. 0.8 or 80%

A system load of 80% means that the system in average is busy 80% of the time and has nothing to do for the remaining 20%. Queues build up during the 80% where the system is busy.

Figure 6.5B assumes one more thing: The service time is 8 seconds *on average*. Shorter service times are frequent and very long service times are possible, but rare. (Technically speaking, we assume that the service time is exponentially distributed with mean 8 seconds. The inter-arrival time is exponentially distributed with mean 10 seconds.)

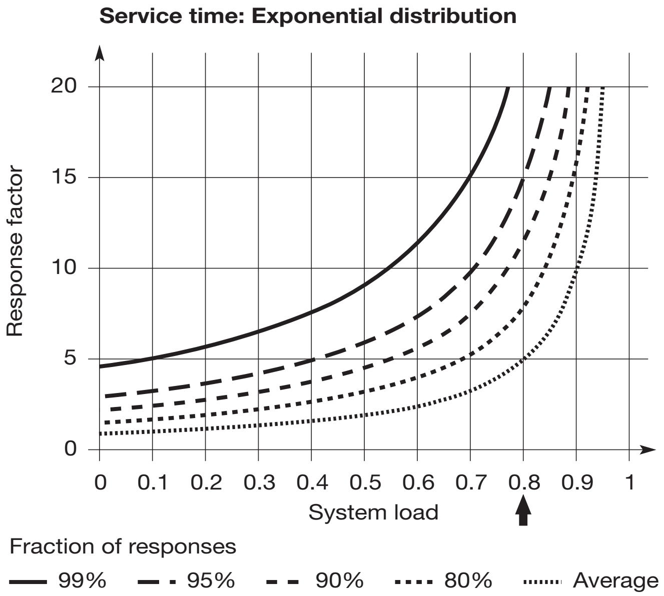
Now look at the graph in Figure 6.5B. The x-axis is the system load or traffic intensity, 0.8 in our case. The y-axis is the response factor, i.e. the response time for a service time of one second.

The vertical line for system load at 0.8 tells us about the response times in our example. Note that the average response factor is 5. Since we have a service time of 8 seconds, the average response time is $5 \times 8 = 40$ seconds. This far exceeds the 20 seconds we wanted in 95% of the cases.

The 0.8 line also shows that the response factor in 90% of the cases will exceed 12. In other words, in 90% of the cases the response time will be above $12 \times 8 = 96$ seconds. Conclusion: We cannot even satisfy the 80 second limit in 90% of the cases.

What should you do if the users really have a request every 10 seconds on average? The answer is to get a faster system, so that the service time is reduced and the system load goes down. Let us try to double the computer speed. We now have these basic data:

Fig 6.5B Response times M/M/1



Example:

Service time: Time to process one request

Average service time:	8 seconds (exponential distribution)
Average inter-arrival time:	10 seconds (exponential distribution)
System load:	8/10 = 0.8
Average response time:	5 × service time = 40 seconds
90% responses within:	12 × service time = 96 seconds

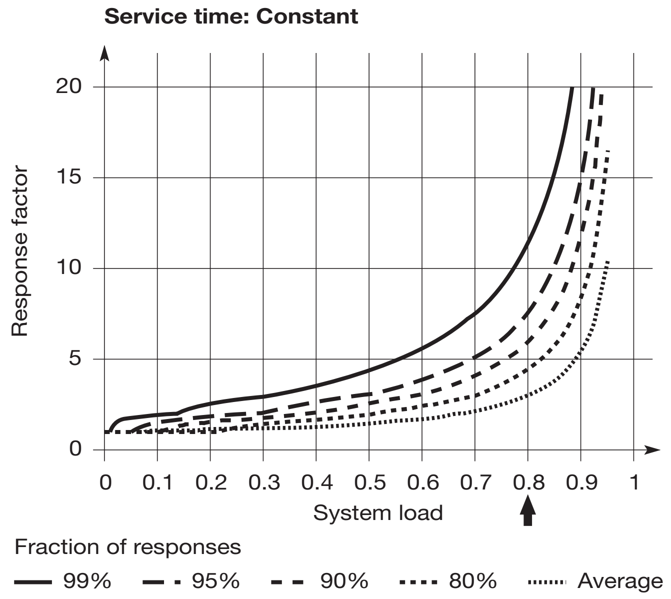
Service time:	4 seconds
Inter-arrival time:	10 seconds
System load:	4/10, i.e. 0.4 or 40%

For this system load, the diagram shows that the 95% response factor is 5 and the response time will thus be below 5 × 4 = 20 seconds in 95% of the cases. This is exactly what we asked for.

The 99% response factor is 8, so in 99% of the cases, the response time is below 8 × 4 = 32 seconds. The customer ought to be happy. The response time will exceed 32 seconds in less than 1% of the cases. It may exceed also 80 seconds, but only very, very rarely.

Warning: If the users, for instance in an emergency, start to access the system more frequently than the estimated 10 second inter-arrival time, the system load will go up and the response time will increase dramatically. We explain more on this below. If the system has to handle such situations, we should specify the inter-arrival times in these periods or specify the response time for a peak burst of events, such as the situation in R2 in Figure 6.5A.

Fig 6.5C Response times, M/D/1



Example:

Service time:	8 seconds
Average inter-arrival time:	10 seconds (exponential distribution)
System load:	8/10 = 0.8
Average response time:	3 × service time = 24 seconds
90% responses within:	6 × service time = 48 seconds

Constant service times

Above we assumed that service times were exponentially distributed, but in many cases they are almost constant, and this decreases response factors. Figure 6.5C shows the response factors when the service time is constant. It also shows the same computation as Figure 6.5B.

Note that the average response time for system load 0.8 is now 24 seconds in contrast to the 40 seconds with exponential service time. In general, the average response factor will be half as long for very high system loads – even though the service time has the same average. For small system loads the average response factor will still be one, of course.

Note also that the 90% response time reduces from 96 to 48 seconds. In general, the 90% curve reduces by a factor two for high system loads. (The other curves behave similarly.)

7.6 Design and programming

If the project includes development from scratch or extensions to a COTS product, design and programming is involved. During these parts of development it is important to ensure that requirements will be met. There are basically three ways to do this:

Direct implementation. Developers implement the system requirement by requirement, in that way tracing requirements to design or code on the fly.

Verification. At suitable moments, developers look at requirements one by one and check that they are met. The check can be done during a design or code review, or the check can involve some kind of testing.

Embedded trace information. For each piece of code or each design artifact, state the requirements IDs that it deals with. This gives you backwards traceability from code to requirements. With a search tool or a requirements tool you can also go the other way and find all the code pieces involved in a requirement, thus getting forwards traceability as well. By going both ways you ensure that each piece of code is justified by some requirement and that each requirement is handled by some code. The forward traceability also helps developers find the program pieces impacted by proposed changes in requirements.

Some requirements are easy to implement one by one, while others are not. Some requirements are easy to verify during development, while others are not. We have discussed this for each requirement style in Chapters 2 to 6.

Embedded trace information sounds easy and useful, but in practice it only works if requirements are easy to implement one by one and easy to verify during development.

Below we will look at the ease of direct implementation and the ease of verification for different kinds of requirements. (Figure 7.6 gives an overview.)

Data requirement

R1 The system shall store data according to this data model . . .

Implementation: direct. Developers can systematically translate each box of the model into a database table.

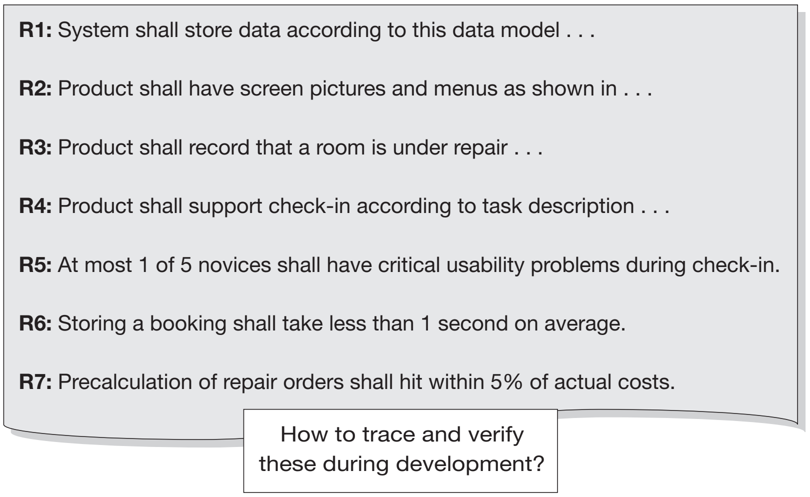
Verification: simple. Other developers can easily check that it is done properly.

Design-level requirement

R2 The product shall have screen pictures and menus as shown in app. xx.

Implementation: direct. Developers can implement the screens and menu points one by one.

Fig 7.6 Design and program



R1: System shall store data according to this data model . . .

R2: Product shall have screen pictures and menus as shown in . . .

R3: Product shall record that a room is under repair . . .

R4: Product shall support check-in according to task description . . .

R5: At most 1 of 5 novices shall have critical usability problems during check-in.

R6: Storing a booking shall take less than 1 second on average.

R7: Precalculation of repair orders shall hit within 5% of actual costs.

How to trace and verify these during development?

Verification: simple. Users can check that it looks right and other developers can review the code to see that it works according to the description. If some of the functions behind the menu points are complex, this is more difficult, of course.

Feature style

R3 The product shall be able to record that a room is occupied for maintenance.

Implementation: modest. The developer has to build a screen for the purpose, or more likely add some functions to another screen. He may care about usability, but cannot do much at this stage.

Verification: simple. Other developers can easily review screens and code to see that it works right.

Task support

R4 The product shall support check-in according to task description *yy*.

Implementation: complex. The developer cannot program this directly. First he has to design the user interface – screens and pictures, then he has to program it. A good design of the screens has to take other user tasks into consideration too.

Verification: simple. Once designed and implemented, it is fairly easy to verify the requirement by carrying out the task – either as a walk-through of which screens to use and what to push, or in a partly working version of the product.

In many cases the user tasks are not visible in the program at all. Well, some developers love to create start screens with main menus labeled *Check-in*, *Create guest*, *Update guest*, etc. The user has to choose one of the possibilities and the system guides him through the rest. In our experience, users are often bewildered by these early choices. Tasks rarely have names that the user will recognize. Tasks are artifacts defined by the analyst. For complex tasks, a good design consists of simple functions that the user can combine in many ways to carry out the complex tasks. The issue is closely related to the many-to-many relationship between domain-level events (the user tasks) and product-level events (the product functions) as discussed in section 3.3.

Usability requirement

R5 At most 1 out of 5 novices shall have critical usability problems during check-in.

Implementation: complex. This requirement is even harder to implement. First the developer has to design the user interface, then usability-test it, then revise the design – probably several times – and then program it.

Verification: done already. The usability test has served as the verification at this stage.

Performance requirement

R6 Storing a booking shall take less than a second on average.

Implementation: complex. The developer may know that this requirement is easy with the usual approach and the given hardware, data volume, and expected traffic. But if he isn’t sure, the requirement is hard to implement. He has to write small test programs to measure response times, adjust indexes, invent improved algorithms, etc. When it looks right, he can make the real program. If it turns out to be impossible, he should report the problem and ask for a requirements change.

Verification: complex. If someone else is going to verify the response time requirement at development time, he has to set up test situations with realistic data volumes and simulated traffic.

Goal-level requirement

R7 Precalculation of repair orders shall hit within 5% of actual costs.

Implementation: complex. This goal is far away from an implementation. It is not even certain which user tasks are involved, and we might have to design new tasks. The developer cannot do this on his own. A large part has to be done by the customer. The analyst should have transferred this business goal to ordinary requirements.

Verification: Should we verify this goal at development time? In principle, no, because a careful analyst would have traced the goal to requirements to ensure that the product can support the goal.

However, even careful analysts make a slip every now and then. A careful developer or Quality Assurance (QA) person should check that the product can support the goal. Have we implemented the necessary screens for capturing and using experience data? Are they easy to use for the staff involved? We should better make a usability test of these critical screens since the customer's most important business goal may be wrecked by low usability. Have we implemented means for management to check regularly how close the company is to the business goal? No, the analysts forgot that, but it is still easy to include. Submit it as a request for change, and remember to give a cost estimate.

The lessons

Many managers believe that developers can deal with one requirement at a time. The developer takes requirement R1 and implements it. Next he takes requirement R2, and so on. In some cases this is true, for instance for the first two or three requirements above.

However, most development doesn't work that way. Development is primarily driven by intuition. Based on a general understanding of the domain and the technical possibilities, the developers come up with designs and program parts. For this reason, it is important that the developers have a good understanding of the domain in order that their intuition works in the right direction. The skilled developer checks regularly that requirements are met. (The bad developer forgets about it, and doesn't realize that there are problems until acceptance testing.)

During development it often turns out that a requirement is impossible to fulfill – at least at a reasonable price. Or developers may notice missing requirements, or wrong requirements that are actually inconvenient to the user. These issues should be handled by requirements management like other changes (section 7.8).

7.7 Acceptance testing and delivery

When the product is delivered, customer and/or supplier test it. Often the product is delivered in several parts, and then each delivery is tested separately. Testing is usually done in several steps, for instance in this way:

Installation test. The purpose of the installation test is to ensure that hardware and software are available for the later tests.

The supplier installs the product (hardware and software as needed) in realistic surroundings and checks that the basic functionality is available. In some projects installation is trivial, while in others it is a huge job. Networks of computers, external products, and special measurement systems may be needed. Depending on what the parties have agreed, the test installation may or may not be at the customer's site.

System test. The purpose of the system test is to check that the product fulfills all requirements that can be verified before daily operation starts. Tacit requirements about correctness, stability, etc. are also tested as far as possible.

The test team use the installed system for the test. In order to carry out the test, they need carefully planned test data and test databases intended to exercise all task variants, capacity limits, etc.

Some requirements are simple to test, others are complex, just as for verification during design and programming. Verification during design and programming may often be done as reviews of the documents, but in a system test, we have to test that things actually work. As an example, during programming we would verify that the system stored the correct data through a review of the database configuration. During system test, we would have to store data and retrieve it, preferably through the user screens.

In the simplest cases the test team have a list of the requirements and work through them one by one to see that the system works as prescribed. For each requirement they write their comments and observed problems on the list. Requirements in the form of task descriptions are particularly easy to handle in this way (see section 7.4).

For more complex requirements, the team has designed a set of test cases, for instance test scenarios, in such a way that each test case covers many requirements. The team keep a careful trace document that shows where each requirement is tested. Then they perform the test cases and write down comments and defects. As an example, testing that the system stores the right data could be done as a side effect of trying some user tasks. The team has to keep track of which database fields have been tested. As another example, response times are tested with a full database, many users working concurrently, etc. The entire setup is used to test many requirements in the same session.

It is more difficult to test tacit requirements, for instance that the system responds correctly to all user mistakes, computes correct results in worst case scenarios with strange data, etc. If something is wrong in this area, the developer can usually see it without consulting the requirements. We are dealing with programming errors, not requirement defects. Developers have several ways to test for programming errors, for instance stress tests and whitebox tests where each program branch is tried out (see Beizer 1990). Areas that are risky from a business viewpoint should be tested more thoroughly than other areas. These matters are outside the scope of this book, however.

If the system contains COTS parts, they will usually not be tested as part of delivery because the supplier will have tested them already as part of product development.

Deployment test. The purpose of the deployment test is to check that the product can work in daily operation with production data.

Supplier and customer use production data for the test, and they create the necessary database files and convert data from the old system. They will not attempt to try out all functions in all variants. The system test has done that, and usually it is not possible to make the tests with production data. However, the team check that the system correctly supports the user tasks, that the real customer records are in the database, that discount codes are correct, that invoices come out correctly on the preprinted forms, that data communication works with real data, etc.

Acceptance test. The purpose of an acceptance test is to check that the product can handle all variants and is installed and ready for daily operation.

An acceptance test is a system test plus a deployment test. These two tests may be performed at different times – as described above – or in combination.

Operational test. The purpose of the operational test is to check requirements that can only be verified after a period of daily operation. It might be response time under daily load, breakdown frequency, task time for experienced users, qualifications of the supplier's hotline, etc.

For each test, the parties decide whether the product has passed. In practice it is a question of whether the remaining problems can be considered maintenance problems to be dealt with later.

Passing the tests determines when the customer pays. Usually the contract states that the customer has to pay most of the bill when the product has passed the acceptance test. When the product also passes the operational test, the customer has to pay the rest.

Usually passing the tests leads to various legal consequences too. Typically, the customer is not allowed to use the system in daily operation until he has accepted that it passed the acceptance test. This rule puts pressure on both parties, helping them to reach consensus without delay. Usually the customer carries the risk for fire and other accidents to the system from the moment the product has passed the installation test.

8.2 Survey of elicitation techniques

Figure 8.2 is a table of useful elicitation techniques. For each technique, we have shown what work products it can produce more or less completely. The darker the shading, the better it is at producing that information.

Fig 8.2 Elicitation techniques

Things to elicit / Techniques	Present work	Present problems	Goals and key issues	Future system ideas	Realistic possibilities	Consequences & risks	Commitment	Conflict resolution	Requirements	Priorities	Completeness
Stakeholder analysis		▒	▓	░		░	▒	▒	░		
(Group) interview	▓	▓	▒	░					░		
Observation	▓	▒							░		▒
Task demo	▓	▓	▒						░		▒
Document studies	▓	▒							░		▒
Questionnaries	▒	▒	░	░							
Brainstorm				▓						░	
Focus groups		▓	▓	▓			▒	░	░	▓	
Domain workshops	▓	▓	▓	▒	▒	▒	▒	▒	▒	▒	
Design workshops			░	▓	▒	░	▒	░	▓		
Prototyping				░	▓	▒	▒		▓		▒
Pilot experiments			░		▓	▓	▓		▒	▒	▒
Similar companies		░	▒	▓	▓	▓			▒		
Ask suppliers		░	░	▒	▓	▒			▒		
Negotiation		░	▒	░	░	░	▓	▓	▒	▓	
Risk analysis			▒		▒	▓			░		
Cost/benefit		▒	▓	░	░	▓		░		▓	
Goal-domain analysis		▒	▓	▒	░	▓	▒	░	▒	▒	▓
Domain-requirement analysis		░	▒	░					▓	▒	▒

Some analysts use the table to find suitable elicitation techniques in specific situations. For instance they may have a list of wild system ideas, and now want to find realistic solutions. Looking down the column of "realistic possibilities", they get several ideas, for instance make a prototype, or go visit some suppliers of that kind of system.

8.2.1 Stakeholder analysis

Stakeholders are the people who are needed to ensure the success of the project, for instance the future daily users and their managers. It is essential to find all the stakeholder groups and find out their interests.

During stakeholder analysis you try to find answers to these questions:

- Who are the stakeholders?
- What goals do they see for the system?
- Why would they like to contribute?
- What risks and costs do they see?
- What kind of solutions and suppliers do they see?

There are various ways to gather this information. You may call a joint meeting where all the known stakeholders are represented, or you may call several smaller meetings. If everything else fails, you have to go and interview stakeholders one by one.

Sections 8.3 and 8.5 explain more about stakeholders and various kinds of business goals.

8.2.2 Interviewing

As Figure 8.2 shows, interviewing is good for getting knowledge about the present work in the domain and the present problems. It is not quite as good at identifying the goals and critical issues, although we may be forced to use it for that in stakeholder analysis. It can elicit some ideas about the future system.

It can give other information, e.g. opinions about what is realistic and where the conflicts may lie, but other techniques are needed to verify the information and resolve the conflicts.

Many analysts consider interviews the main elicitation technique, and it can of course be used for many things, depending on whom you ask and what kind of questions you ask. The limitation is the elicitation problems we listed above (see also B.G. Davis 1982).

Whom should you interview to get information about current work and current problems? Preferably some members from each user group. Make sure that you interview not only the officially nominated representatives of the user group, but other staff members as well. Often management has nominated a representative (typically a middle-level manager) for a group of users, but experience shows that many representatives don't really know what is going on in the daily business, although they believe they know. Getting information from the real end users can be critical.

What should you ask about? That depends upon when you ask. Initially, you would ask broad questions about day-to-day work, day-to-day problems, and other items on the list of things to elicit. Make sure to ask about *critical tasks*. When does the user work under stress? When is it highly important that things are 100% correct? You can rarely identify these tasks from observation: you have to ask for them. See more about task identification in section 3.10.

You should also try to find out why these tasks are carried out. Sometimes, users are uneasy at being asked such a question, because they cannot really explain properly. If you ask a manager, he may even be directly offensive: I am the manager, don't question why I do as I do. A good idea is to ask *when do you do this* instead of why. People don't feel offended at that. (Thanks to Andrew Gabb for this trick.)

Later, when you have identified critical issues, you ask more detailed questions, e.g. about data volumes, task times, detailed work procedures.

Always prepare yourself by writing a list of questions to ask. Leave ample space at each point for notes during the interview. You don't have to follow the list point by point during the interview. Try to follow the interviewee instead, and consult your list every now and then. Be open to new issues that turn up during the interview, but don't let yourself get sidetracked. I always write my planned list at the upper half of the page with space for notes, and leave the lower half blank for unforeseen issues.

Instead of individual interviews, you can conduct group interviews. A group of users from the same work area can tell you more about the present work, problems, and critical issues than individual interviews. The group inspire each other to remember critical issues, describe day-to-day work, etc. An important thing when conducting such interviews is to keep a balance between participants so that nobody can dominate, and all feel safe at giving their opinion.

8.2.3 Observation

Users are not always aware of what they really do and how they do it. If you ask them, they may come up with logical, but wrong explanations, not because they are lying deliberately – they are doing their best.

As an example, consider a simple task. How do you find a section in a cookbook, manual, or textbook that you know well? Like most people, you would probably say that you use the list of contents or the index. Observation, however, shows that in 80% of cases where this would be a good approach, people start skimming through the book, believing that they can remember or guess where the section is. Only if that approach doesn't succeed do they use the logical approach with indexes, etc.

One way around this mental blindness is to observe what is really going on. The analyst can spend some time with the users, observing their daily tasks. In some cases, analysts use video cameras (with the users' permission) to lengthen the period of observation. This has the advantage that you can later review the tapes with the users and ask what really happened.

Observation vastly improves your knowledge of the current work and some of the associated work problems. It also serves as a check of other information. Unfortunately, the real critical issues and tasks often escape observation. With a power distribution system, for instance, the critical situation is when something goes wrong once a year. The analyst is rarely around at that critical moment.

8.2.4 Task demonstration

A variant of interviewing and observation is task demonstration. You ask the users to show you how they perform a specific task.

In many cases the users cannot explain what they do in their daily work. But they are able to show you how they do specific tasks. Task demonstration is also a way to observe rare, critical tasks.

As an example, an analyst asked a public servant what tasks he had and how he handled them. He couldn't explain, he said, because the tasks were so different. All right, said the analyst, what are you going to do as the next thing today? That was easy to explain, and the user could also explain how he was going to handle that task. What would be the next job, the analyst asked. Oh, something very similar, the user replied. This is quite typical – when asked, the user found it difficult to describe a typical task, but when explaining specific examples, he saw the similarities.

The analyst was also interested in finding out how things had been done before the present computer system was introduced, but the user had not been able to explain this. Now the analyst asked how this particular task would have been done before; and the user could readily explain this to him.

Usability problems. Sometimes you want to identify usability problems in the present system, for instance in order to document how the new system could improve user performance. Mostly, experienced users are not aware of any problems, although there may be serious ones. To identify the problems you need some kind of task demonstration.

The best way in this case is to run a usability test where users carry out frequent or critical tasks by means of the existing computer system. You or a usability specialist observe what the users do, note the time used, errors made, number of keystrokes, etc. You may also ask the users to think aloud so that you understand what they really try to achieve or why they make certain mistakes (see sections 6.6.2 and 10.5). Later you analyze the problems and document the potential for improvement.

8.2.5 Document studies

Document studies are another way to cross-check the interview information. It is also a fast way to get information about data in the old “database”.

The analyst studies existing documents such as forms, letter files, computer logs and documentation of the existing computer system. He may also print screen dumps from the existing system.

8.2.6 Questionnaires

The above techniques get information from relatively few users or stakeholders. You can use questionnaires to get information from many people. You can use them in two ways: to get statistical evidence for an assumption, or to gather opinions and suggestions.

In the first case you ask closed questions such as, “How easy is it to get customer statistics with the present system: very difficult, quite difficult, easy, very easy…” You can use the results to see how important the problem really is.

In the second case, you ask open questions. Essentially, you can ask the same questions as during an interview, e.g. “What are the three biggest problems in your daily work?” and, “What are your suggestions for better IT support of your daily work?” It can be quite difficult, however, to interpret the results. The respondents might have misunderstood your questions, and you may misunderstand their answers. During an interview, you can check your understanding immediately and ask questions you hadn’t thought of before.

It is difficult to classify the results of open questions since you cannot clearly see whether two issues really are the same. For this reason, open questions should be asked of relatively few people.

There is also a high risk of misunderstanding with closed questions. To reduce this risk, it is essential that you know the domain already. Interviews are a good way to start.

Before sending a questionnaire form to many people, always test the form on a few people from the target group. You will be surprised how much they can misunderstand you. Revise the form and make another test before you send out the final version.

8.2.7 Brainstorming

In a brainstorming session you gather together a group of people, create a stimulating and focused atmosphere, and let people come up with ideas without risk of being ridiculed. The facilitator notes down all ideas on a whiteboard. Soon each idea spawns new ideas, most of them ordinary ideas, some stupid, and some very promising. An important rule of the game is not to criticize any idea. Even seemingly stupid ideas may turn out to have a valuable “diamond” seed in them.

During elicitation, the focus is on goals and requirements for the new system. If creativity doesn't come by itself, the analyst may raise a few issues he has noticed during interviews. Later, during system design, brainstorming sessions focus on innovative ways to meet requirements.

As a result of not criticizing the ideas generated, there will be many unrealistic ideas. The facilitator may finish the session with a joint round where participants prioritize the ideas. Some facilitators insist on not prioritizing at the meeting. They know that if you sleep on it for a couple of nights, some stupid ideas may turn out to be brilliant. If prioritized right after the meeting, they might have been killed.

8.2.8 Focus groups

Focus groups resemble brainstorming sessions, but are more structured. The term *future workshop* is also used to mean roughly the same. A focus group starts with a phase where participants come up with problems in the current way of doing things. Next comes a phase where participants try to imagine an ideal way of doing things. The group also tries to explain why the ideas are good. That helps formulate goals and requirements for the new system.

Several groups of stakeholders should participate, and at the end of the session, each group identifies their high priority issues. When later prioritizing the requirements, it is important that each stakeholder group gets solutions to some of their high-priority issues. If a stakeholder group doesn't get anything in return, they are rarely willing to contribute to the system.

Focus groups also create an excellent understanding between stakeholder groups and often a joint commitment to succeed.

Section 8.4 gives details of how to run a focus group.

8.2.9 Domain workshops

There are many kinds of workshops and the term blends into brainstorming sessions and prototyping. At a workshop users and developers co-operate to analyze or design something.

Here we will describe two types of workshops: domain workshops where the team map the business processes, and design workshops where the team design the user interface.

The result of a domain workshop may be task descriptions, dataflow diagrams, or activity diagrams that describe what goes on in the domain. Later, the analysts turn the descriptions into requirements. The path to requirements is short if task descriptions are used as requirements, but long if requirements have to be in the traditional feature form.

As a side effect of the workshop, the team may specify system goals and critical issues.

It is important that expert users participate in domain workshops. They know all the business details in their own domain. Sometimes, however, expert users know only their own narrow work area and lack an overview of the bigger picture. To gain this overview, experts from several work areas may have to participate, and the analyst will have a fascinating job in trying to make ends meet. Things are much easier if you can find experts with cross-domain expertise.

Managers may participate, but they rarely know the real details of the procedures and cannot replace the expert users. However, they may be instrumental in defining goals and visions.

8.2.10 Design workshops

At a design workshop users and developers co-operate to design something, usually the user interface. The term "co-operative design" means roughly the same.

This form of workshop is widely advocated, but often the result is a disaster. The reason for this is that the users become so absorbed by design and technical issues that they turn into developers and become very committed to the solution they design.

Why is that a problem? Because, in their enthusiasm, they may forget whether all business goals and tasks are covered, and whether other users back in the organization understand the solution.

If you use this kind of workshop, it is crucial that the team every now and then checks the user interface against the tasks descriptions and the business goals (see section 7.6). The team should also usability-test the design with users who have not participated in the design process.

8.2.11 Prototyping

A prototype is a simplified version of part of the final system. Developers experiment with the prototype to get an idea of how it would work in real life. The result of the experiment can be two kinds of requirements:

Product-level requirements. The experiment has shown that the required functionality is realistic (feasible) and useful. The requirements can for instance be stated in feature style or as a task description with an example solution. It is *not* a requirement that the real system has an interface like the prototype.

Design-level requirement. The real product shall have an interface exactly like the prototype. The experiment has shown that such an interface satisfies the goals of the system.

An important kind of prototype is a simplified version of the user interface. The system needs no or little functionality. The facilitator simulates the rest during the experiments. For instance, the system could consist of empty screen pictures and a simple mechanism that allows the facilitator to select the next screen. The facilitator simulates system output by means of yellow stickers attached to the screen and filled out by hand.

If the prototype has been usability-tested against real tasks, it can become a design-style requirement. If not, it is only an example of what the interface might look like. The requirement itself has to be less design-oriented (e.g. features or task descriptions).

Other system parts can be prototyped too. As an example, the product might have to communicate with another, existing product. An experiment where a prototype communicates with the existing product about something in the domain can reveal a lot. What are realistic response times? Is the expected functionality actually present in the existing product? Can we use the results for our tasks?

8.2.12 Pilot experiments

In many cases, the new system will be COTS-based, perhaps with some added functionality. The cost of the system may be high, but the main risk is whether the organization can adapt to the system and use it to improve performance. The organizational changes themselves are often more costly than the product.

In this situation, much of the risk can be eliminated through a pilot experiment. A small part of the organization tries the new system on a trial basis, but with real production data. At the same time they experiment with changed work procedures. The project team observes the results and evaluates the cost and benefits of the new system. Usually they also suggest different ways to use the system if it comes to large-scale deployment.

If the experiment succeeds, it creates a high degree of commitment. It also helps to identify the final requirements and their priorities. The main problem is that the system must be operational to a large extent.

It would be wonderful if we could run pilot experiments with prototypes, and some developers report that they actually do so, but we are not too sure whether they really do this. Maybe they run an extended prototype test where several users play roles during the test – without using the system for production work. Anyway, more experiments in this area are important.

8.2.13 Study similar companies

One of the best sources of realistic ideas is to see what other companies do to handle problems similar to your own. A study of their procedures and comparison with your own can give you many ideas. They may also have experience with the specific product you are considering. Most importantly, a visit to their site makes it easier to imagine how the new system could work.

Aren't other companies reluctant to share such knowledge? Yes, it may happen, particularly if they are competitors, but often the study is mutually beneficial, and they are willing to share experiences.

There are other ways to get information about competitors' procedures. Some international auditing and consultancy companies have a huge benchmark database with performance figures for other companies in your field. Performance is measured for many kinds of internal processes such as recruitment, internal IT support, etc. At least you can find out how your performance compares to others. Improvement is possible if you are not in the top ten – and the consultancy company might give you a clue about how to do this – for a fee, of course.

8.2.14 Ask suppliers

Suppliers of the products you are considering are also an important source of ideas for new solutions. Aren't they just trying to sell their own stuff? Yes, but often they also know a great deal about how others use their product, and they may refer you to some of their customers.

They can also give you a long list of features they provide. Often the customer realizes that he has been much too modest in his initial requirements. State of the art offers much more than he dreamt about.

If you compare features from several potential suppliers, you may realize that many of your hard-derived requirements are useless: every supplier can satisfy them. What makes a difference are two things: the quality requirements (e.g. efficiency and ease of use) and the special features that you haven't thought about (see also Maiden and Ncube 1998).

In this case you should stop worrying about the standard functionality, but pay attention to the quality requirements. You should also consider the special features that you didn't think about initially. If your team can come up with innovative ways of using the special features, you should specify them as requirements.

8.2.15 Negotiation

The purpose of negotiation is to *resolve conflicts*. It can be a matter of conflicts between supplier and customer, but more serious conflicts often arise between various stakeholders inside the customer organization.

Conflicts between supplier and customer are usually discussions about costs, benefits, and who runs the risk. Conflicts inside the customer organization can have many other agendas, e.g. power struggles and conflicts with other projects about resources.

A group discussion with conflict resolution on the agenda has participants from the conflicting parties. To be fruitful, the parties must be willing to talk together and try to understand each other. Otherwise, preparatory work should be done on an individual basis, or the whole issue escalated to a higher level in the organization. (The threat of escalation can often bring the parties to a constructive meeting.)

There are many tricks available for resolving conflicts, for instance to have each party explain what they believe the other party wants and why.

From a requirements point of view, the most important thing is to analyze the goals for each party. Often conflicts are about the solutions, though everybody can accept each other's goals. The trick is to find solutions that don't conflict, but support everybody's goals (a win-win situation). Section 10.3 explains a case where the proper technical solution resolved a national conflict.

8.2.16 Risk analysis

Risk issues play different roles during elicitation and development. The purpose of risk analysis during elicitation is to identify risky areas of the project and find ways to reduce the risk. In the elicitation stage you primarily look at the possible consequences to work procedures, client relations, the customer's IT staff, etc.

You can identify the risks by working with stakeholders. Ask how the work in their area should proceed after deployment of the new system. What kind of changes are needed, and what are the risks that these changes are blocked? Which potential conflicts do they see with other stakeholders? Then try to find ways to reduce the risks, for instance by involving and motivating users, planning training and change procedures, conducting experiments, and defining additional product requirements.

As in other cases, you can work with stakeholders on an individual basis or in joint meetings and workshops. The difficult part is to imagine the future work situations. If stakeholders cannot do that, there is a great risk that the work situation may be unacceptable. It might help to do as with task demonstrations: take specific examples of work cases and imagine how they would be carried out. Or even better, use a mockup system to simulate that you carry them out.

Later, during transition to development, you do risk assessment and risk management. You look at risks related to requirements and technical issues, e.g. can the supplier develop what is required, what happens if a sub-contractor doesn't deliver what you expect? You can manage the risks in various ways. This is a big area in itself. We give an introduction in section 9.3.1. You can read more in Boehm (1989) and Jones (1994).

8.2.17 Cost/benefit analysis

A cost/benefit analysis looks at the entire project and compares the costs of doing it with the benefits resulting from it. Traditionally, costs and benefits are expressed in money terms, and many analysts rightly claim that it is impossible to measure all the relevant factors in such terms.

However, there is also a broader meaning of cost/benefit. When we say cost/benefit we mean both hard factors (money terms) and soft factors (qualities). The terms tangible and intangible benefits mean the same.

Examples of hard factors are: changed revenue, changed costs, product costs, training costs.

Examples of soft factors are: customer satisfaction, employee satisfaction, decision quality, reaction time to external changes. Section 8.6 shows a detailed example.

8.2.18 Goal-domain analysis

A goal-domain analysis looks at the relation between business goals and tasks (or other domain issues).

We might call it a checking technique, but it is an important part of elicitation since it can drastically change requirements. In many cases important goals are forgotten during elicitation. No requirements deal with them, and as a result the final system doesn't meet the goals. You may also see the opposite: a feature or task that doesn't seem to have a goal. The feature may be superfluous or a goal may be missing. Section 8.7 shows ways to deal with this.

8.2.19 Domain-requirements analysis

A domain-requirements analysis is similar to a goal-domain analysis, but works at a lower level. For instance, we could have an issue saying

the system shall be easy to use.

Since this is not a verifiable requirement, we have to do something about it. In many cases we can translate it into good requirements. Section 8.8 shows the principle and a simple example. Section 10.4 gives a complex example.