

Software Requirements Styles and Techniques

SOREN LAUESEN



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 pre-calculated 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.

Fig 1.6C Recommendation: why and how

Measuring neural response can be painful for the patient. Electrodes must be kept in place . . . so both hands should be at the patient during a measurement.	Domain – why
R1: It shall be possible to perform the commands <i>start, stop</i> . . . with both hands at the patient.	Requirement
Might be done with mini-keyboard (wrist keys), foot pedal, voice recognition, etc.	Example – how

Recommendations

In general it is an advantage to include “business” goals, domain-descriptions and – using more caution – examples.

Business goals and goal-level requirements are not requirements to the supplier, but:

- 1 they improve the supplier’s understanding of the domain, and
- 2 they allow you to check that the goals are fully reflected in the requirements and that they are considered during development.

Domain-descriptions are sometimes requirements, sometimes not. If you use product-level requirements, make sure that you add some domain description to explain the *purpose* and the context of the requirements. Domain descriptions:

- 1 improve the supplier’s understanding of the domain,
- 2 help the supplier to guess tacit requirements or ask for further information,
- 3 take additional space unless you use domain-level requirements.

Examples of design are not requirements, but:

- 1 they improve the supplier’s understanding of what the customer had in mind,
- 2 they record early ideas,
- 3 they should be used with caution to avoid requirements issues turning into design issues.

1.7 Typical project models

Highlights

Product-level requirements: elicited from users. A lot of work.

Domain-level requirements: describe user tasks. Much faster.

Two-step approach: domain-level first, design-level later with careful checks.

Contracts and price structure: can reduce risks.

In the preceding sections we have mentioned many possibilities: requirements on various levels from goal-level to design-level; different requirements styles from plain text to diagrams, and from data requirements to quality requirements; many techniques and methods from elicitation and analysis, to validation and verification.

The rest of the book gives details of all these possibilities, but how do you combine them in practice? There is no fixed, standard way to do it, because projects are so different. What is excellent in one project may be a waste of time in another. In spite of this, we will explain a couple of models below. You may use them as a basis for your project and adjust them according to actual project characteristics. Figures 1.7A and 1.7B give an overview of the possibilities.

1.7.1 The traditional approach: product-level requirements

In this classical model you interview the stakeholders, study existing documents, conduct workshops and brainstorms. Finally you analyze and write the requirements specification. In the basic version of the model you don't need to specify business goals, because the decision to get a new system has already been made. In the extended version, you also conduct interviews and brainstorming sessions to identify business goals.

The specification will contain the following central parts:

- Introductory parts (including business goals in the extended version).
- The limits of the system: what is to be delivered; what are the interfaces (for instance shown as a context diagram; see section 3.2).
- Data requirements, for instance as a data model with a data dictionary (see sections 2.2 and 2.3).
- Product-level functional requirements, for instance as function lists, feature requirements, and textual process descriptions (see sections 3.3, 3.4, and 4.3).
- Quality requirements for critical quality factors (see section 6.2).

Fig 1.7A Typical project models

Traditional: Product-level requirements

Ask users, study documents, extract features.

- Introduction, [business goals]
- System limits, e.g. context diagram
- Data requirements, e.g. data model, data description
- **Product-level functional requirements, e.g. features**
- Critical quality requirements

Fast approach: Domain-level

Describe user tasks, study documents . . .

- Introduction, [business goals, **BPR tasks**] . . .
- System limits, e.g. context diagram
- Data requirements, e.g. verbal data description
- **Domain-level requirements, e.g. Tasks & Support**
- [Trace analysis: **goals to tasks**]
- Critical quality requirements

Two-step approach: Domain-level + design-level

- All the fast-approach stuff
- +
- **Design-level requirements, e.g. prototypes, protocols**

The IEEE standard 830 gives additional guidance on how to organize each part, particularly the introductory matters and the functional requirements. The Volere Template (Robertson and Robertson 1999) has a much longer list of things to consider in the specification.

After the first draft of the specification, it is necessary to review it, have the customer validate it, check it for consistency, and so on. Chapter 9 describes ways of doing this.

The work may take a long time, and often the analyst has to do most of it himself. Usually there is insufficient time to do it right and get information from all the stakeholders. Furthermore, analysts tend to focus on the trivial parts, leaving the difficult parts till later when – alas – time runs out.

In some cases, the work is primarily done by users and co-ordinated by the customer's IT department, for instance as follows: the IT team asks all user departments involved to submit their requirements to the new system. The departments have no requirements expertise, but they try to come up with something. The IT team then collects and edits the submissions and sends them to the suppliers.

Fig 1.7B Recommended project models

Project type	Project model		
	Trad.	Domain	Two-step
In-house	?	OK	OK
Product development	?		OK
Time and materials	?	OK	OK
COTS business COTS tools	? OK	OK	
Tender COTS Tender tailor	? ?	OK OK*	OK*
Contract COTS Contract tailor	? ?	OK OK*	OK*
Sub-contract	OK		OK
Maintenance	OK		

? Used, but dubious * Variable price

In the West Zealand Hospital case, explained in section 3.8, the IT teams worked in this co-ordinating way. They didn't fully understand what the users submitted, but assumed that the supplier would understand it. The supplier actually understood some of it because he had domain expertise from many other hospitals, but unfortunately he didn't understand many parts. "We will find out more during the implementation process," he reasoned and signed the contract. No wonder that the system later failed to meet several business goals.

Apart from the long time spent in analysis, the traditional approach has some inherent weaknesses. In general, it is difficult to ensure that product-level requirements provide adequate task support and that they satisfy the business goals. Furthermore, even if customers can read the requirements, they will not know whether they are adequate. If you invite stakeholders to submit requirements, they also tend to come up with long wish lists that are difficult to prioritize afterwards.

Product-level requirements are well suited to three kinds of projects, as shown in Figure 1.7B:

- 1 COTS purchases where the customer wants a development tool or a technical component. The requirements are a list of desired features, and the different products are compared against the list.
- 2 Sub-contracting where customer and supplier agree on a technical sub-product and its features.
- 3 Maintenance projects where an existing, deployed system gets new features.

In all of these cases, the customer knows the user tasks and the supplier doesn't need to know about them. Furthermore, the best type of solution has been decided. What remains to be found is the exact list of features needed.

1.7.2 The fast approach: domain-level requirements

In this model you primarily focus on describing the user tasks in co-operation with expert users. You also collect information on the data to be stored in the computer. In the basic version of the model you don't need to specify business goals, because the decision to get a new system has already been made. In the extended version, you also conduct interviews and brainstorming sessions to identify business goals. Part of describing the business goals may be to describe high-level tasks or BPR (Business Process Re-engineering) tasks.

The specification will have almost the same central parts as the traditional approach, with important deviations shown in bold:

- Introductory parts (including business goals and **BPR tasks** in the extended version; see section 3.11).
- The limits of the system: what is to be delivered; what are the interfaces (for instance shown as a context diagram; see section 3.2).
- Data requirements, for instance as a data dictionary (see section 2.4). Sometimes a data model is useful too.
- **Description of user tasks** plus a requirement saying that these tasks have to be supported (see sections 3.6 and 3.8).
- A **trace analysis** showing how each business goal is reflected in the tasks (extended version only). See section 8.7 for an example.
- Quality requirements for critical quality factors (see section 6.2).

Before sending the specification for review, you conduct a CRUD check (Create-Read-Update-Delete check, section 9.2.3) to see that all required data are handled in some task. Usually they are not, so then you need to describe the missing tasks.

Next you send the specification out for review and conduct review meetings. Stakeholders are primarily invited to correct the task descriptions. They may also suggest solutions, but these are examples and not requirements.

Chapter 15 shows an example of this approach in a small, low-cost project expected to take only a few weeks.

In section 3.8 we explain how this approach worked in a hospital case, and we contrast it with the traditional approach above. The analysts reported that the domain-level approach reduced analysis hours and calendar time by a factor of 5 to 10. Selecting the right product was speeded up in a similar way.

The weakness of the approach is that some functional requirements are hard to specify through user tasks. It is of course acceptable to specify them in other ways as a supplement.

The approach is suitable for most types of projects, with a few exceptions:

- 1 It is not suitable for COTS acquisition of development tools or technical components. In these cases the customer wants certain features, and the domain aspects have already been resolved.
- 2 It is not suitable for sub-contracting of technical parts for the same reasons.
- 3 It is only suitable for product development as the first step of a two-step approach. In product development we strongly recommend that you design the user interface and test it carefully as the second step of requirements.

1.7.3 The two-step approach: domain-level requirements plus design-level requirements

The first step of this model is the domain-level approach above. It produces the first level of requirements exactly as above with business goals, data requirements, task descriptions, etc. Reviews and checks are also carried out.

The second step is a design task that produces *design-level requirements* for the complex interfaces.

For the user interface, developers design a prototype of the screens. The basis is the task descriptions and the data description. Section 2.5 outlines a systematic way to do it. The design is usability tested to see that the user interface is sufficiently easy to learn and effectively supports the tasks (see section 6.6). The result is a new requirement on design level:

- The product shall have a user interface with screens, menus, etc. as shown in app. xx.

Review and validation of this requirement has been done by means of the usability test.

For complex technical interfaces, developers design and specify communication formats, protocols, etc. (see section 5.5). It is also a good idea to make a functional prototype that tests whether the communication really works.

In principle the design-level requirements replace the domain-level requirements from step 1. Verification at the end of development is now a test to see that design-level requirements are fulfilled, e.g. that all parts of the user interface prototype now work as specified. However, don't throw away the requirements from step 1 just yet. Let us see why.

Requirements management. During design, developers detect many errors and omissions in the domain-level requirements. For instance, they may detect missing user tasks or new task variants. Should they update the domain-level requirements when they are no longer needed for verification? Yes, they should! There are several reasons for this. First, the team needs the domain-level requirements to validate the design-level requirements. Second, the domain-level requirements are excellent for requirements management. They serve as justification for the design-level requirements and show what the consequences would be of a change.

Finally, in practice it is also useful to verify the domain-level requirements at the end of development, including testing that the user interface is still easy to use. The reason for this is that deviations may sneak in during development and make the user interface less useful. A typical example of this is a creative Web designer, who adds fancy colors to the final Web page, unintentionally making it hard to read on the screen.

The two-step model is suited to many types of projects:

- 1 In-house development, time-and-material based development, and sub-contracted projects.
- 2 Product development. We have seen repeated success with the two-step model in product development where programming, etc. became much more predictable, client satisfaction sky-rocketed, and sales prices could be doubled (Lauesen and Vinter 2001).
- 3 Tailor-made systems, either in tender projects or contract development.
- 4 The add-on parts of COTS-based products, either in tender projects or contract development.

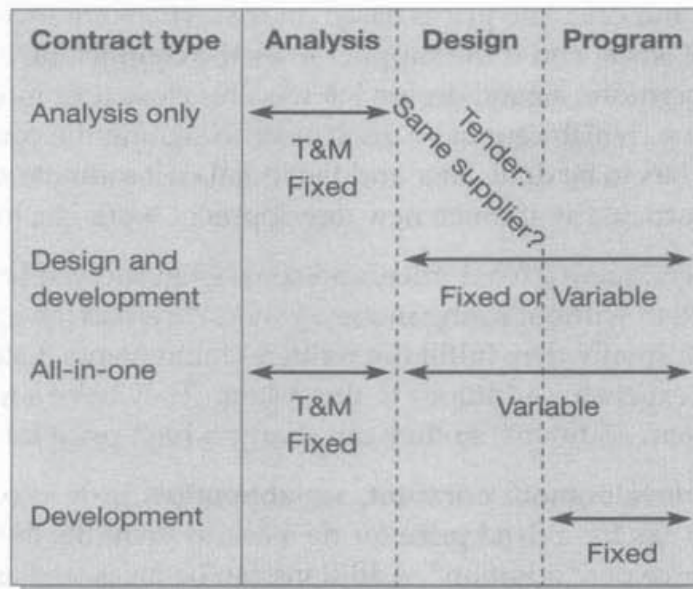
In 3 and 4 above, a contract is involved and it is important to find a fair way to specify the costs. See the discussion below.

The two-step model is not suited to the standard parts of COTS-based systems. They already have their own interfaces, so defining new ones is meaningless.

1.7.4 Contracts and price structure

When you use the two-step approach in tenders or contract development, deciding on a final price can be tricky because the size of the project is only partly known at the beginning of step 2, the design step. Do you write one contract for the design work and another for the development work? Or could you write one contract for the whole thing? Figure 1.7C shows an overview of the various possibilities.

Fig 1.7C Contracts and prices



Initial analysis contract. You may draw up a contract for the initial analysis that ends with domain-level requirements (step 1). Most consultants prefer to do this on a time-and-material basis.

Some consultants offer a fixed-price contract for the initial analysis. This may sound risky, since the amount of work involved is unknown. However, from an initial investigation of the customer's situation, consultants can often tell roughly how much time will be needed – and how much the customer is willing to pay.

The result should be the necessary requirements and other material for a tender process or a contract negotiation. The price in the contract may either be fixed or variable, as explained below.

Some consultants are also suppliers of tailor-made solutions or COTS-based solutions. In this case a consultant may not be allowed to submit proposals in the tender process. However, if the requirements are truly domain-oriented, all potential suppliers have equal opportunities and the consultant who wrote the requirements can also qualify as a supplier.

In contrast, if the requirements are more design-oriented, the consultant will most likely favor a particular supplier, typically himself, and he is disqualified since he has a de facto monopoly. There are two ways out of this: (1) Avoid the design-oriented requirements in the tender and use only domain-oriented requirements. (2) Avoid the tender situation and go for an all-in-one contract that includes analysis and development (see below).

Design and development contract, fixed price. When responding to a domain-level specification with a fixed price, suppliers will essentially have to do some design work and calculate prices based on that. There are several drawbacks here: this is a lot of work, and if the supplier loses the contract, he has wasted a lot of money. Furthermore, a good design job requires close user interaction, usability tests, and so on, which cannot be done prior to signing the contract. So the real design work has to be done later and the initial cost estimate may be very inaccurate, particularly if much new development work has to be done.

Many suppliers quote a fixed price, and some even succeed in delivering an adequate system without losing money. Others do a bad job and lose money initially, but formally they fulfill the written requirements. Later they recover the loss through expensive additions to the system. They have a monopoly when it comes to system additions, so they can charge a high price for these.

Design and development contract, variable price. Instead of a fixed price, the customer can ask for a fixed price for the basic system (the built-in COTS parts) plus a unit price per "addition". Additions can be measured as the number of new screens graduated according to screen complexity (sections 3.16 and 5.1 gives examples). Additions can also be measured by means of Function Points or Object Points, which are standardized ways of measuring the essential size of an IT-system (see Thayer 1997 and Furey and Kitchenham 1997).

This type of contract often specifies the development approach to be used, for instance how users should be involved, how to review and test designs, etc. The supplier should include these costs in the fixed price he quotes. The advantage to the supplier of this is that writing a proposal will be much less work, and at the same time he reduces his risk because more work later means more money.

The advantage to the customer is that he doesn't end up paying a high price for additions when the supplier has got a monopoly. He can compare suppliers' prices, but to do this he must have some idea of what the supplier provides as a standard solution and what has to be added. The Midland Hospital requirements in Chapter 12 show a way of doing this based on product features. The Task & Support style shown in section 3.8 and in the West Zealand Hospital case (Chapter 13) show how to do this with domain-oriented requirements.

In practice we see amazingly few contracts made with variable prices. For one reason or another, the parties prefer a fixed price in spite of the associated risks. Sometimes the parties add a clause saying that either party, up to a certain point in development, can withdraw from the contract against paying compensation to the other party. This is a good way out if the supplier turns out to be less competent, or the customer more demanding, than expected.

All-in-one contract. An all-in-one contract includes analysis and development. This form is widely used in cases where the customer has little IT expertise, doesn't want an expensive consultant, and feels confident that a supplier can provide what he

needs. The supplier will carry out the analysis, design the solution and develop the product. It is unwise to do this on a time-and-material basis, but which price should be agreed on initially, and what are the requirements? The real requirements develop over time, so the only requirement the parties should agree on initially is the development process to be used. For instance, they could state in the contract that a two-step approach is to be used with details described further (see section 3.16).

The contract might be a combination of an initial analysis contract and a design-and-development contract with variable price. The price structure could be composed accordingly: fixed price or time-and-materials basis for the initial analysis; fixed price plus a unit price per “addition” for the rest of development.

Contract outlines and price structures of this type also allow the customer to compare several suppliers.

Development contract, fixed price. If the customer has used design-level requirements, it is fairly simple to draw up a contract for the rest of development. The risk to the developer is small, since it is clear what has to be developed. The customer has taken the risk that his design is adequate.

We see this type of contract in industry, for example, to develop the software for a mobile phone where the customer has designed the exact user interface of the phone. It is also common in aviation and the aerospace industry. Since the system may change over time, it is also a good idea to contract on a unit price per “addition” to prevent the supplier from exploiting his monopoly later. In most of these cases, the customer is a main contractor.