Software requirements

Styles and techniques

Soren Lauesen

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

Preface

Have you ever used a new piece of software that didn't meet your expectations? If so, it might be because nobody stated the expectations in a tangible manner. Software requirements are about writing the right expectations in the right way.

These days, many people get involved in writing requirements. It is not only a job for specialists; users, customers, suppliers, and programmers also get involved. In small companies we sometimes even see employees without special training being asked to write requirements for a new software product. Furthermore, the roles of expert user, analyst, designer, and programmer seem to blend more and more. This book is important and relevant for many people involved in software requirements:

The analyst, working as a requirements engineer or a consultant, can find tricks here and there, and he can look at requirements written by other specialists.

The customer can find ways to ensure that the new product will meet his business goals, and suggestions for handling contracts and tenders.

Software suppliers can find ideas for helping the customer and for writing competitive proposals.

Users can prepare themselves for working with specialists or the developers. They can also find ways to describe their work tasks, and examples of what to write and what not to write in their requirements.

Programmers and other **developers** can learn how to express requirements without specifying technical details, and how to reduce risks when developing a system.

IT students can learn about theory and practice in requirements engineering, and get a foundation for case studies and projects.

You don't have to read the whole book. How can we cover so many topics for so many audiences? The answer is simple: you don't have to read all of the book. If you read most of Chapter 1, you should then be able to read sections of the book in almost any order, according to your needs.

Background

When I began to work in the software industry in 1962, software requirements were relatively unimportant since at the time hardware was very expensive, and software was comparatively cheap. Renting a computer for an hour cost the same as paying someone to work for 30 hours and computers were 5000 times slower than they are today.

Software development was carried out either on a time and materials basis, or as a small part of the really important job – making better hardware. The customer paid until he had a program that printed results he could use with some effort. Nobody thought of usability. Everything to do with computers was a specialist's job.

Today things have completely changed. Hardware is cheap, and software development is expensive and very hard to keep within budget – particularly if the customer wants a result matching his expectations. For this reason software requirements are growing in importance as a means for the customer to know in advance what solution he will get and at what cost.

Unfortunately, software requirements are still a fuzzy area. Little guidance is available for the practitioner, although several textbooks exist. One particularly critical issue is the lack of real-life examples of requirements specifications.

This textbook is based on real-life examples, and it discusses the many ways of specifying software requirements in practice. We emphasize practical issues such as:

- what analysts write in real-life specifications; what works and what doesn't work
- the balance between giving the customer what he needs and over-specifying the requirements
- the balance between completeness and understandability
- the balance between describing what goes on in the application domain and what goes on in the computer
- reducing the risk to both customer and supplier
- writing requirements so that they can be verified and validated
- writing low-cost requirements specifications.

During my time in industry, I have worked as a programmer, a project manager, and later as a department manager and quality manager. However, I always loved programming and had a key role in the critical parts of the programs. We programmed many things, from business applications, scientific applications, and process control, to compilers, operating systems, and distributed databases.

When I worked as a developer from the mid-1970s, our team had to write software requirements, but we always felt uncertain about what we wrote. Was it

requirements or design specifications? We realized that requirements were important, but felt stupid not knowing what to do about it. Furthermore, nobody else in our multinational company could show us a good example of software requirements, although there were corporate rules and guidelines for what to write.

In the mid-1980s I became a full professor in software engineering at Copenhagen Business School. That let me see development from two other sides: the user side and the customer side. I didn't have the constant pressure of turning out code and products, so I had time to look at the industry from another perspective.

For a long period I studied human–computer interaction and came up with systematic ways of developing good user interfaces – the missing links between studying the users and producing a good prototype. To my disappointment, industry didn't care at that time (the Web has now changed that attitude).

In the early 1990s, I decided that it was time to change subject. I asked around in industry to find out what was the most difficult part of development. Everyone I asked said "requirements and all that stuff at the beginning of the project." That was how I became interested in requirements.

I went to my research advisor, Jon Turner of New York University, and said, "Jon, I want to do research in requirements." He looked at me for some seconds and said, "Don't." "Why?" I asked. He replied that it was impossible to do anything significant in that area, and what researchers actually did had little to do with what industry needed. Alan M. Davis (1992) has observed the same thing.

This was a real challenge to me. To begin with, I had great problems in getting to see other people's requirements. I talked to developers from many companies and asked them: "Do you write software requirements?" Usually they said yes. I then asked, "Could I see the one you are using or writing right now?" There was a pause – then various replies, such as, "No, it's confidential, and it would be too much trouble to get permission for you to read it." Or, "Well, it isn't quite finished yet; maybe you could see it later." Or even this amazing variant, "Well, we're working on it, but right now we are too busy testing the system. When we have finished testing, we will write the requirements, and then you may see them."

Every now and then I got permission to see some real-life software requirements. Usually they were inspired by the IEEE 830 guidelines, since they contained all the introductory sessions such as Scope and Audience. However, when it came to the specific requirements, they were bewildering, and IEEE 830 suggested no guidance. Part of what I saw was program design; there were also some dataflow diagrams, and the rest made little sense to me. Where were the requirements?

Six months later, I saw some software requirements that were so good that I could learn from them. Jens-Peder Vium was the first to show me a good requirements specification, and it is included in this book as the Danish Shipyard case (see

Chapter 11). Although vastly better than anything else I had seen at that time, it too had deficiencies, and together we worked on improving the various techniques involved. Soon my studies gained momentum, and I got to see many other good requirements, some of which are included in this book. A year later, so many people wanted me to look at their requirements that I had to say no to many of them.

My conclusion from these initial studies was that people were ashamed of the requirements they had written, but they didn't know how to make them better. Furthermore, everybody had some good parts in their specification, and some serious weaknesses. If all the good things could be combined, we would be close to a general solution. However, there were some important problems that none of the practitioners seemed able to solve:

- How do you avoid writing anything about the product, yet be able to verify its requirements?
- How do you ensure that the requirements correctly reflect the customer's business goals?
- How do you specify quality factors such as usability or maintainability in a verifiable manner?

Research, experiments, and luck helped me develop answers to these questions. These answers are included throughout the book, for instance in sections 3.8, 6.6, 6.11, and 8.7.

Using the book for courses

The book is a considerably extended version of an earlier book, which we used successfully at professional courses for analysts and developers, as well as for computer science students. Depending on the audience, we selected different parts of the book for discussion. We have even used the book with Information Systems (IS) students with no understanding of programming. In this case we combined it with a short course in data modeling, data flow, and basic understanding of development activities.

The figures in the book are available in PowerPoint format, and the checklists as Word documents. Solutions to some of the exercises are available for teachers. E-mail the author at slauesen@itu.dk. Most of the figures are rich in detail, and as a result, you can easily spend 5–30 minutes discussing a single figure. In a typical course, only about one-third of the figures are discussed.

The book suggests two kinds of course activities, *discussions* and *exercises*. Discussions are themes for course room discussions, and may also be used for homework. Exercises are for homework or for teamwork during course hours.

Exercises and training projects

You can run the exercises in many ways. At professional courses, we assign exercises to teams of three to five participants. Each team has to outline the answer in one to two overheads. That should be possible in about an hour, depending on the participant's background and level of knowledge.

For university students, the exercises are given as homework, but here too we tend to restrict answers to a few overheads. One or two teams present their solution to the other students. About 15 minutes are allowed for a presentation, including discussion. The students are asked to control the presentation themselves. They should usually imagine that they are developers or consultants, while the other students are "customers". It is important to listen to the "customer", explain the solution again if the customer hasn't understood it, and identify weaknesses in one's own solution. A successful presentation identifies many weaknesses. This attitude is extremely important in practice, but difficult to achieve because we all tend to defend our own solutions.

However, exercises alone are not sufficient for training in requirements engineering. While programming exercises may give you programming training, this is not so with requirements. The art of discovering real demands and stating real requirements cannot be practiced through written exercises.

It is necessary to practice using real companies. For university courses, we always combine the course with the students doing project work in a real company. The first part of the project is that the students have to find a company or organization on their own. This also trains them to find the way to the right people; a very important skill in requirements engineering.

Acknowledgements

This book has only one author, yet I mostly write "we" in the text. This is because most of the experiences I discuss and report here have originated in talks and collaboration with someone else. Thus a large and varied selection of my colleagues have contributed to the book and justify my use of "we".

I would particularly like to thank the following:

Jens-Peder Vium, of Innovation & Quality Management, for permission to use the Danish Shipyard case (Chapter 11), and for many inspiring discussions and joint presentations. He has been a consultant for many years, and is an important source of knowledge about many different kinds of projects.

Susan Willumsen, at that time a masters student, for her collaboration and sharp observations during the Danish Shipyard study.

Houman Younessi, of Swinburne University, now Rensselaer at Hartford, for many theoretical and practical discussions that were the starting point of this book, and for some of the ideas behind the style concept and the maintainability requirements.

Otto Vinter, of Bruel & Kjaer, for permission to use part of the Noise Source Location requirements (Chapter 14), some of the case studies, and for many inspiring discussions, particularly about error sources and prevention methods.

Karin Lomborg (now Karin Berg) of Deloitte & Touche, for permission to use part of the Midland Hospital case (Chapter 12).

Jan C. Clausen, of Katalyse, for helping me to see the basic difference between tasks and use cases, and for many inspiring discussions about requirements and usability.

Klaus Jul Jeppesen, of Asea Brown Boveri, now the IT University, for information about large projects in control and manufacturing, customer negotiations, etc.

Marianne Mathiassen, masters student, and Lotte Riberholt Andersen, Jeanette Andersen, and Annemarie Raahauge, of West Zealand county, for collaboration when developing the technique first known as 'use cases with solutions', later renamed to Tasks & Support.

Lene Funder Andersen, Lene Frydenberg, Jens Wolf Frandsen, and Marc Olivier Collignon, diploma students, for being the first to try Tasks & Support in real life. They successfully managed to use the technique for writing requirements and run the tender process for a large telecommunications company. They also helped the company select the right proposal from among twenty suppliers.

Dorte Olesen, Lars Henrik Søfren, and Jette M. Rosbæk, of West Zealand county, for their impressive work when trying out Tasks & Support in a new hospital project.

Erik Simmons, Intel Corporation, for teaching me Planguage and for reviewing the book as carefully as if it had been a requirements document. (Like a typical developer, I couldn't repair all the defects. ☺)

Soren Lauesen
June 2001

slauesen@itu.dk

1

Introduction and basic concepts

A requirements specification is a document that describes what a system should do. It is often part of a contract between a customer and a supplier, but it is used in other situations as well, for instance in in-house development where "customer" and "supplier" are departments within the same company.

Specifying requirements is recognized as one of the most difficult, yet important areas of systems development. Little guidance is available for the practitioner, although several textbooks exist. One particularly critical issue is the lack of real-life examples of requirements specifications.

The structure of the book

We have based this book on real-life examples and in it we discuss the many ways of specifying requirements. It contains a requirements specification for a total business administration system in a shipyard (Chapter 11), a short, but adequate specification for a membership administration system (Chapter 15), excerpts from two hospital systems (Chapters 12 to 13), and excerpts from a 3D sound measuring system (Chapter 14).

After the introductory chapter, the book has five long chapters on various ways to state requirements, illustrated by examples. Most readers don't understand why this comes so early. They find it more logical to start with elicitation techniques, i.e. techniques for gathering requirements, such as interviews, prototypes, and brainstorms. Elicitation is needed before any requirements can be defined, so why not start that way?

The reason is a simple observation: if the analyst doesn't know how requirements can be stated in real life, he cannot elicit them. He may ask the customer a lot of

1.3 Contents of the specification

Highlights

Data and functional requirements.
Quality requirements: system speed, ease of use, etc.
Floating transition to contract issues.
Parts to help the reader: business goals, diagrams, etc.

What should the requirements specification contain? Theoretically it is simple: it should specify the input to the system and the output for each input. In principle, that is what the user will see in the final system, so nothing else needs to be specified. (Theory says that the specification should also state how fast the system shall produce this output, and other performance matters.)

This is the traditional idea in computer science, and scientists have developed many ways of specifying input and output in exact detail. In order to write the specification, the analyst must elicit the requirements by studying users and business goals. The assumption, however, is that the end result is a specification of input and output.

Unfortunately, real-life systems are usually too complex to specify in this way, so it is necessary to specify them on a higher level. Furthermore, it is not a simple matter to derive precise specifications that ensure good user support and meet the business goals.

Let us look at the situation in more detail. Figure 1.3 shows the system as a black-box with *interfaces* to the surroundings. First of all, the system has user interfaces to various user groups. It also has interfaces to the supporting hardware and software platforms, for instance commercial products such as Pentium PCs, Windows NT, Oracle databases, and SAS (for data analysis).

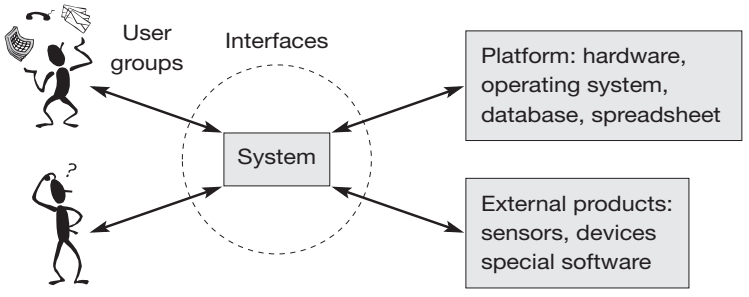
The system may also have interfaces to external technical systems, for instance special sensors and devices, public data-bases, document-handling systems, natural language processors, and so on. The diagram is actually a *context diagram* – a very useful part of requirements explained in section 3.2.

Data requirements

An important part of the requirements is *data requirements*: What data should the system input and output, and what data should the system store internally?

Database. Most systems have to keep track of information about their surroundings, for instance customers, or valves and temperatures in a chemical

Fig 1.3 Contents of ReqSpec



Data requirements:
System state: Database, comm. states
Input/output formats

Functional requirements, each interface:
Record, compute, transform, transmit
Theory: F(input, state) -> (output, state)
Function list, pseudocode, activity diagram
Screen prototype, support tasks xx to yy

Quality reqs:	**Managerial reqs:**
Performance	Delivery time
Usability	Legal
Maintainability	Development process
. . .	. . .

Other deliverables:	**Helping the reader:**
Documentation	Business goals
Install, convert, train . . .	Definitions
	Diagrams . . .

plant. The system has to store the corresponding data in some kind of database or other internal objects. It is important to specify this data, and in Chapter 2 we explain various ways to do this. Database data is independent of the interfaces and is most conveniently described in a separate section.

Some analysts claim that these data details are internal computer matters and should not be specified in requirements. However, there is almost a one-to-one relationship between information found in the surrounding domain, and the data stored in the system. As a result, specifying the data has very little to do with designing the system.

Input/output formats. Input and output data appear on the various interfaces. The data requirements should in principle specify the detailed data formats for each interface, but in practice many details are specified indirectly through the database

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.

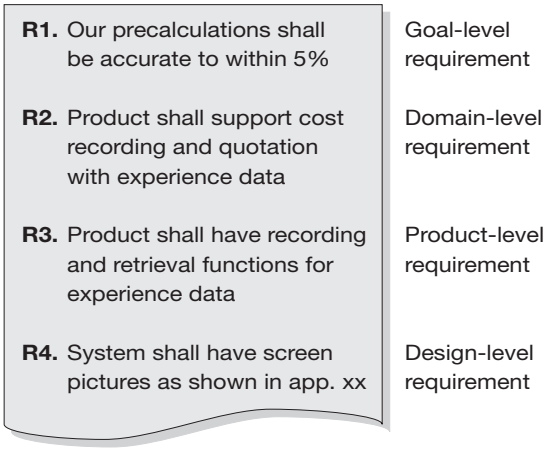
When preparing a quote, the sales staff precalculate the costs, but often the actual costs exceed the precalculation, causing the shipyard to lose money. Or the pre-calculated cost is unnecessarily high, causing the shipyard to lose the order. What is the solution to this? Maybe the new IT system could 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



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.

3.8 Tasks & Support

What is it?

Structured text describing tasks, domain problems, and possible support for them.
Identifies critical issues.
Discusses product features in a structured way.
Easy to understand for user as well as developer.
Easy specification of variants and complexity.
Simple to verify.
Domain-level requirements – also suited to COTS.

Plain task descriptions as discussed in section 3.6 are *domain-level models* of the activities, in the sense that we only explain what human and computer do together. We don't even distinguish between how we did the task in the old days and how we want to do it in future. We don't require a specific solution, but leave that to the supplier or developer, as long as the solution supports the user tasks.

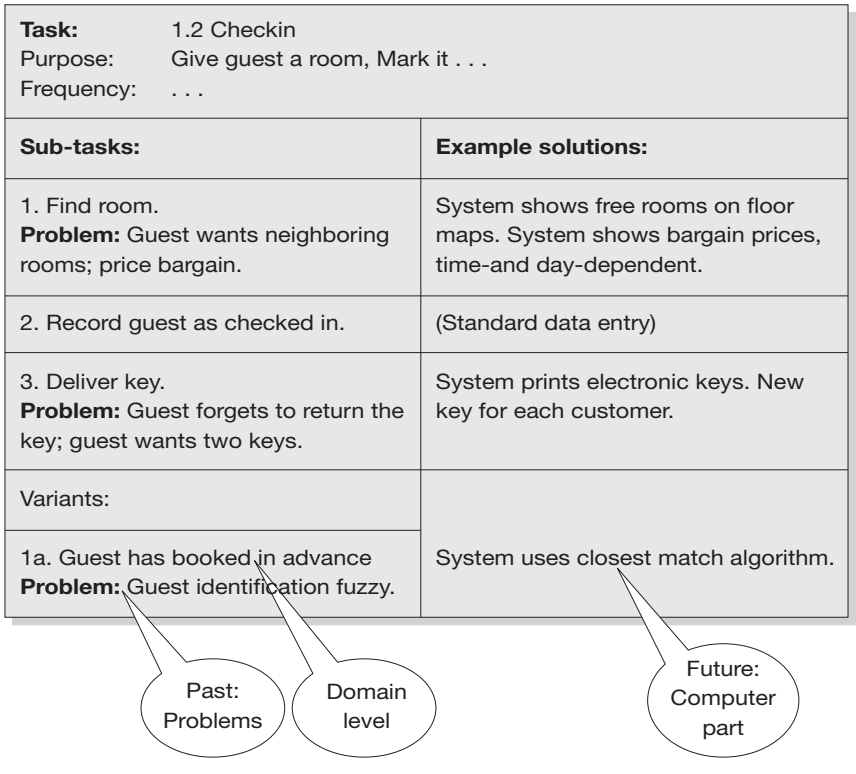
However, the customer often wants some influence on the solution or he may have suggestions for solutions. Understandably, he is tempted to specify product features. On the other hand, the supplier may not be able to provide those features at a reasonable price – or he may have better solutions than those envisaged by the customer.

Tasks & Support resolve this dilemma. Figure 3.8A shows how we could use this style to specify the check-in task. Here are the differences from plain task descriptions:

- Each sub-task is described in two columns.
- **Domain-level.** The left column explains the domain-level activity, i.e. what human and computer should do together.
- **Problems.** The left column also explains any issues or problems in the old way of doing things.
- **Solution.** The right column describes a possible solution that could support the sub-task. Supplier and customer may later co-operate to specify another solution.
- **Example vs. agreed.** The heading of the right column changes during the process, for instance from *Example solution* to *Proposed solution* to *Agreed solution*.

The Task & Support idea was developed by this author and Marianne Mathiassen in close co-operation with a large customer (a hospital) and three COTS suppliers (Lauesen and Mathiassen 1999). Section 10.7 has more information on the case study. The technique has since been used successfully in several large projects, both by vendors, customers, and product developers.

Fig 3.8A Tasks & Support



Requirements

What are the requirements with this approach? There are two options:

R1 The product shall support tasks 1.1 to 1.5 and remedy the specified problems.

R2 The product shall provide the features in the right-hand column of tasks 1.1 to 1.5.

The first possibility corresponds to plain task descriptions, although with emphasis on issues and problems. The solutions are just examples. This is usually the best choice since the supplier has to ensure adequate task support.

The second possibility corresponds to *Features from task descriptions*. It is a good choice at a later stage when the parties have agreed on the way the tasks must be supported. It is useful to preserve R1 as a requirement to ensure task support also in matters not covered by R2.

6.7 Usability requirements

Highlights

Many ways to measure usability.
Some ways are suitable for new product parts, others for choosing COTS.
Some ways are risky to all parties.

Usability can be specified and measured in many ways. Figure 6.7 shows nine styles for usability requirements. For each style, we have indicated the risk to the customer and the supplier when using the style. The risk to the customer is that although he may get what is specified, he may not get what he really needs. The risk to the supplier is that he may not be able to meet the requirements – or only with excessive costs. Below follow the details of each of the styles.

Problem counts

R1 At most one of five novices shall encounter critical usability problems during tasks Q and R. The total list of usability problems shall contain at most five medium problems. (Critical and medium problems are defined in 6.6.1.)

In a hotel system, tasks Q and R might be booking and checking in. The requirement covers ease of learning quite well. We might include other tasks to cover the system better. If the user is an experienced receptionist (i.e. he has experience from another system), he might also give us an impression of the task efficiency and the ease of understanding.

We should specify more precisely what we mean by *novice*. Novice concerning reception work and/or novice concerning this particular product? We should also specify how much instruction they have been given. (In small hotels there are often temporary staff and night receptionists who get at most 10 minutes of instruction from a more experienced staff member.)

The great advantage of this style is that the requirement can be tested early. For the COTS parts of the system, we can carry out the test before signing the contract. For a tailor-made system, developers can test the requirement during design, and this test is at the same time a natural part of good development. If the requirements are based on task descriptions and variants, they provide excellent test cases (see section 3.6).

The biggest problem with the style is that it is very dangerous to the supplier in the case of a tailor-made product. With the present state of the art in usability, it is hard to know whether the requirement is feasible at all. Another problem is that we are less sure of catching the essence of usability. As an example, we get only indirect indications of task efficiency and subjective satisfaction.

Fig 6.7 Usability requirements

	Risk: Customer	Risk: Supplier
Problem counts **R1:** At most 1 of 5 novices shall encounter critical problems during tasks Q and R. At most 5 medium problems on the list.		
Task time **R2:** Novice users shall perform tasks Q and R in 15 minutes. Experienced users complete tasks Q, R, S in 2 minutes.		
Keystroke counts **R3:** Recording breakfast shall be possible with 5 keystrokes per guest. No mouse.		
Opinion poll **R4:** 80% of users shall find system easy to learn. 60% shall recommend system to others.		
Score for understanding **R5:** Show 5 users 10 common error messages, e.g. *Amount too large*. Ask for the cause. 80% of the answers shall be correct.		
Design-level requirements **R6:** System shall use screen pictures in app. *xx*, buttons work as app. *yy*.		
Product-level requirements **R7:** For all code fields, user shall be able to select value from drop-down list.		
Guideline adherence **R8:** System shall follow style guide *zz*. Menus shall have at most three levels.		
Development process requirements **R9:** Three prototype versions shall be made and usability-tested during design.		

Figure 6.7 shows these problems as a large gray box for the supplier, indicating a large risk for tailor-made parts, and a smaller dark box for the customer, indicating inadequate coverage of all the usability factors.

Task time

R2 Novice users shall be able to perform tasks Q and R in 15 minutes. Experienced users shall be able to perform tasks Q, R, and S in 2 minutes.

This requirement style explicitly covers ease of learning and task efficiency. We can verify the requirement through usability tests. Task efficiency, however, is hard to verify until we have some experienced users to test with.

It is even harder to verify the requirement during development. We need a functional prototype, since mockups give a false picture of the speed. Further, we cannot use think-aloud tests because thinking aloud slows the user down. As a result developers get too little feedback to improve the design.

COTS parts. Surprisingly, the style is low-risk to both parties for the COTS parts of a product. Why? Those parts are finished and the measurements can be made before the buy decision. It may even be possible to find experienced users in another company, in that way measuring also task times for experienced users.

Defining the proper time limits is a problem, of course. However, the open target approach (section 6.3) is suited because the supplier may tell you what is achievable. In many cases, the supplier knows how well his product fares without having to make new measurements.

Tailor-made parts. For tailor-made parts, the style is still excellent from the customer's viewpoint because it can cover important aspects of usability. However, the style is very risky to the supplier. It is not certain that the requirements can be met at all. Furthermore, they cannot be assessed early in development because a functional prototype is needed.

Figure 6.7 shows this as a very large gray box for the supplier, indicating a huge risk for tailor-made parts, and no box for the customer indicating that if he gets what he specified, usability is well covered.

Keystroke counts

R3 Recording breakfast shall be possible with 5 keystrokes per guest, and without using the mouse.

This requirement style covers efficiency for experienced users. If we also require certain response times from the system, we are able to calculate the total task time. We can calculate the user time of the task by means of existing measurements of how fast an average user can press a key, move a mouse, etc. (the *keystroke-level model*, see section

10.6). We further have to add the time for the user to get the data from clients, think about the results, and so on, but this is largely independent of the user interface.

The big advantage of this style is that we can check the requirement early in development. We don't even need access to real users. As a result, the supplier has virtually no risk.

The disadvantage is that we cannot be sure that users find out how to do it in the efficient way, although training may help, of course. Further, this kind of requirement doesn't attempt to cover ease of learning, understandability, etc. If we add usability requirements written in some of the other styles, we can cover these missing points, and still check all the requirements during development.

Opinion poll

R4 80% of users shall find the system easy to learn and efficient for daily use. 60% shall state that they would recommend it to others.

With this requirement style, we ask users about their opinion, typically with questionnaires using a Likert scale. This covers the usability factor *subjective satisfaction*, and it is tempting to believe that it catches the essence of usability.

Unfortunately, users often express satisfaction with their system in spite of evidence that the system is inconvenient and wastes a lot of user time, causes erroneous transactions that IT staff have to deal with, etc. (If the manager knew about this, he would not be as satisfied as the users.)

Satisfaction with the system is heavily influenced by organizational factors, which the supplier cannot control. Another problem with the subjective style is that it is hard to verify the requirement during development. Many usability experts ask users about their subjective opinion after prototype-based usability tests, but it may not correlate well with opinions after system deployment.

The result is that both customer and supplier run a high risk.

Score for understanding

R5 Show 5 users 10 common error messages, for instance

> Amount too large [when drawing money from an ATM]
> Ask them what the cause might be. 80% shall give the correct answer.

This requirement shows a way to measure *understandability*, in the example error messages from an Automatic Teller Machine. Some subjective assessment of the correctness of the answer may be necessary, but ask a teacher to run the test. He will be able to mark the answer as A, B, C, or D. The requirement could be that 80% get A or B.