Søren Lauesen

Software Requirements
Styles and Techniques

This is a sample of some pages from the book.
The table of contents reflects the whole thing.

Samfundslitteratur

Contents

Preface

When I started in the software industry in 1962, we didn't talk about software requirements. At that time hardware was very expensive, and software was comparatively cheap. Renting a computer for one hour cost the same as 30 man-hours. And computers were 5000 times slower than today.

We did all software development either on a time and materials basis, or as a small part of the really important thing: making hardware. The customer just paid until he had got a program that printed results he could use with some effort. Needless to say, usability was of no concern.

Today things have changed completely. 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 have had a growing importance as a means for the customer to know in advance what 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.

I have based this textbook on real-life examples and discuss the many ways of specifying requirements in practice. Researchers tend to think of requirements in an idealistic way, stressing issues like completeness, formal correctness, unambiguity, etc. The book emphasizes practical issues such as

- what analysts write in real-life specs, what works and what doesn't,
- the balance between giving the customer what he needs and over-specifying the requirements,
- the balance between completeness and under-standability,
- 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 (we share this point with the researchers).

Audience
The textbook is aimed at two audiences, system developers and IT students with some background in systems development. Although we discuss classical issues such as data modeling and dataflow, the book is not intended to teach you these disciplines.

We have successfully used the book at professional courses for experienced developers as well as computer science students. We have even used it with IS students with no understanding of programming. In the latter case we combine it with a short course in data modeling, dataflow, and basic knowledge of development activities.

Using the book at courses
The book has a layout designed for courses. On most right-hand pages you find an overhead slide (reduced size). On the opposite side you find the "textbook part" - an explanation of the concepts and techniques mentioned on the slide. (There are many exceptions to this pattern, of course.)

During lectures and discussions, students add their own notes besides the overhead. After lectures, they use the textbook part for homework. Or rather, the bright students do. Admittedly, many students never read the text part, what becomes evident at exams when they only know the contents of the *slides*.

Most slides are quite rich in detail. As a result, you may spend 5 to 30 minutes on a single slide, explaining what it says and discussing it with the students.

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

Slides and suggested solutions to discussions as well as exercises are available for teachers.

Exercises and training projects
You can run the exercises in many ways. Usually, we assign exercises to teams of three to four students. Each team has to outline the answer in one to two overheads. That should be possible in about an hour, depending on the student's general background, of course.

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 imagine that they are developers or consultants, while the other students are "customers". It is important to listen to the "customer", explain if the customer didn't understand, and identify weaknesses in one's own solution. The more weaknesses identified, the better the presentation. This attitude is extremely important in practice, but

quite difficult because we all tend to defend what we have done.

However, exercises alone are not sufficient for training in requirements engineering. While programming exercises may give you programming training, not so with requirements. The art of discovering the real demands and stating real requirements cannot be practiced through written exercises. The author would have to write what the students would have to discover. And then there is no discovery!

Practice in real companies is necessary. For students, we always combine the course with project work in a real company. One part of the project is that the students have to find a company or organization on their own. This also trains them in finding the way to the right people, a very important skill in requirements engineering.

Comments to the first edition
This first edition of the book lacks several things. The wish list for the next edition includes issues such as

1. Handling of goals, e.g. QFD and cost/benefit.
2. Object-oriented matters in detail.
3. Communication protocols.
4. Entity-function matrix.
5. State-transition matrix and composite states.
6. Mathematical specifications.
7. The key-stroke model for usability requirements.
8. Security requirements.
9. Requirements management.
10. Sources of requirements-related errors in practice.
11. Standards for requirements.
12. Improved English, including an English version of app. B.
13. The origin and research behind various techniques.

1. Introduction & Basic Concepts

A requirements specification is a document which 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.

Structure of the Book
We have based this book on real-life examples and we discuss the many ways of specifying requirements. A full real-life requirements specification is found in app. A, excerpts of two other specifications in app. B and C.

Most readers are puzzled why the book starts with examples of requirements. They find it more logical to start with elicitation techniques, i.e. techniques for gathering requirements. (Interviews, prototypes, and brainstorms are examples of elicitation techniques.) 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 questions but not the crucial ones. Or he may use prototyping in situations where it doesn't make sense.

So after a brief explanation of the role of requirements and what a specification typically contains, we give examples of requirements specified in different ways (*styles*).

Styles
A requirement can be specified in many *styles*. We may specify it in plain text, as diagrams, or as tables. We may specify what the software system must do or what kind of user activity it must support. These possibilities may be combined in many ways, giving rise to many styles.

Techniques
With an understanding of the styles, we introduce the various techniques for elicitation, checking, and validation.

Some techniques and notations are explained further in chapter 6 on *detailed techniques*.

The many styles and techniques are options for practical use, not something you have to do in every project.

Many practitioners ask why all these styles and techniques are necessary. Why cannot we just present the right ones that always lead to the good result? I wish we could – give all of you the silver bullet that kills any monster. But a long life in the IT industry has taught me that projects are different. What is crucial in one project, may be waste of time in another. The best advice is to become a good analyst:

The good analyst knows many techniques, but also when to use them and when not.

He combines and modifies techniques according to specific needs.

The aim of this book is to help you on that path. Teach you many techniques, and advice you on when to use them and when not.

1.1 Role of Requirements

A requirements specification plays several roles during a project. Initially it is made in close collaboration between customer and supplier (or their representatives), later it is used by developers. If things go wrong, the requirements specification may end up in court as evidence of what was agreed.

1.1.1 Requirements During Development

The requirements specification is produced early in system development. The slide shows the traditional waterfall model for systems development. In this idealistic model, development starts with an *analysis* phase that produces the requirements specification. The specification serves as input to the design phase, which in turn produces input to programming; and so on, ending with operation and maintenance of the system.

The waterfall model represents an ideal. Real proj??ects don't work like that. Developers don't complete one phase before starting on the next. Often they realize that something went wrong in an earlier phase, so part of it has to be redone. Sometimes it is an advantage to do some design work during analysis in order to produce a better requirements specification. Sometimes developers also make iterative analysis, design, and even programming.

All this means that the phase idea of the waterfall model is wrong. The activities are not carried out sequentially and it is unrealistic to insist that they should. On the other hand, developers analyze, design, and program. The activities exist, but they are carried out in an interleaved and iterative fashion.

This has several implications for requirements. First of all, it is necessary to check that the result of development matches the requirements. This is called *verification* and is shown on the slide as requirements flowing into the test activity. Ideally, verification should be done several times during development to check that for instance the design matches requirements.

Second, requirements will change during development as developers and customers find missing, wrong, and unrealistic requirements. This is called *requirements management*. Requirements management is also important during maintenance, since most changes are caused by changing requirements.

1.1.2 Requirements Elicitation

Where do the requirements come from? In principle, they come from users and other *stakeholders* of the system. The role of analysis is to *elicit* the requirements from the stakeholders.

The assumption is that stakeholders have some *demands* and the role of the *requirements engineer* is to elicit these demands and formulate them as requirements. The people eliciting and formulating requirements are called requirements engineers by some people, *analysts* by others. We will use both terms.

In practice, the analysts can be developers, expert users (preferably a team of both), independent consultants, etc.

Elicitation is a very difficult process for many reasons:

- Stakeholders cannot express their needs, or they ask for a specific solution that does not meet their real needs.
- Stakeholders can have conflicting demands.
- Demands often change over time. For instance, when the users see a smart system somewhere, they realize that they need something similar themselves.
- Users find it difficult to imagine new ways of doing things or the consequences of things they ask for. When they for instance see the system built for them, they often realize that it does not fulfill their expectations, although it fulfills the written requirements.

Furthermore, even when users can express their needs, requirements engineers find it difficult to write them down in a precise way.

The result is that the real demands and the written requirements do not match completely. When a demand is not reflected in the requirements specification, we say it is a *tacit requirement*. We also talk about *tacit demands*, demands that the user is not aware of or cannot express.

For all these reasons, it is necessary that stakeholders check that development meets their real demands. This kind of checking is called *validation*. The slide shows it as information flowing from development to stakeholders. Validation is typically performed on the written requirements specification and at the end of development (*acceptance testing*). Validation during development is an advantage too.

Stakeholders Demands Elicitation

Tacit demands & reqs

Analysis

Validation

Reqspec

Design

Verification

Program

Test

Op & maint
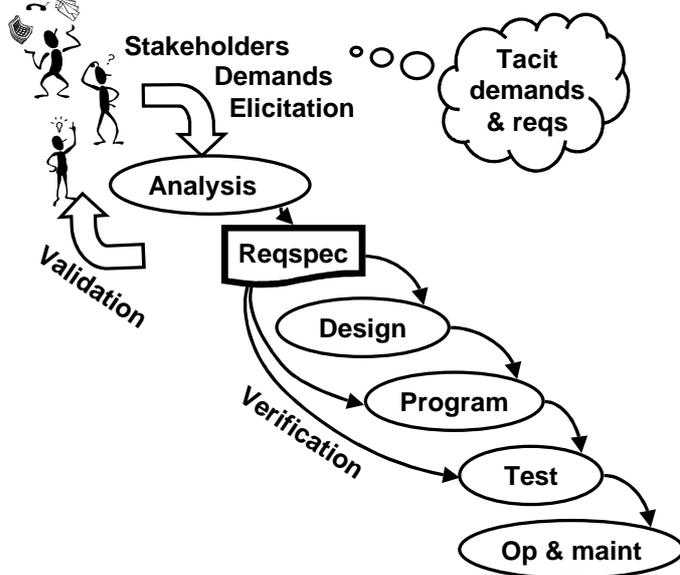
Project types	Customer	Supplier
In-house	User dept.	IT dept.
Prod. devel.	Marketing	SW dept.
Time & materials	Company	SW house
Contract devel.	Company	SW house
Tender	Company	Supplier
Sub-contracting	Supplier	SW house

1.1.3 Project Types

Requirements play somewhat different roles in different types of projects. Here are some typical projects:

In-house development. A project carried out inside a company for its own use. The customer and the supplier are departments of the same company.

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

Cost-based contract. A software house develops the system on a time and materials base. Requirements are informal and develop over time.

Tender. The customer company makes a tender pro??cess where various suppliers are asked to submit proposals. The tender documentation contains an elaborate requirements specification. The customer writes the specification or asks a consultant to do it.

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. Signing the contract is often the last step of a tender process.

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.

1.2 Contents of Requirements Specs

What should the requirements specification contain? The basic idea is simple: It should specify the input to and output from the system. In principle, that is what the user will see in the final system, so nothing else needs to be specified.

This is the traditional idea in computer science, and scientists have developed many ways of specifying input and output in detail. Unfortunately, real-life systems are usually too complex to specify in this way, so it is often necessary to specify them on a higher level.

Let us look at the situation in more detail. The slide shows the system as a black-box with *interfaces* to the surroundings. In the example, the system has user interfaces to the various user groups and interfaces to the supporting HW and SW platform, e.g. Pentium PC with Windows-NT, Oracle database, and SAS for showing complex results.

The system can interface to other systems as well, for instance special sensors and devices, public databases, special software such as a document handling systems, natural language processors, etc.

1.2.1 Functional Requirements
The *functional requirements* specify the input and output for each interface. This can be done in many ways, as we will see later.

According to the computer science ideal, the functional requirements specify three aspects:

Input and output. The format of input and output data and the possible sequences of data.

System state. What data the system stores. The stored data determines the system's state. The system output depends not only on the system input, but also on the system state.

Process (transformation). How the system transforms input to output. The input also causes the system state to change.

What is the system state? You may know the state concept from state-transition diagrams. From a programmer's point of view, the state is the value of the variables, including the entire database. In pragmatic terms: When you give the system a command, the response depends not only on the command, but also on the contents of the database and other variables.

Specifying the functionality according to the ideal above for a complex system is no easy matter, and in

practice it is only done for particularly difficult system parts. Instead developers rely on more informal descriptions, leaving the details to design and programming.

1.2.2 Non-functional Requirements
The *non-functional requirements* specify how well the system performs its intended functions. Non-functional requirements are also called *quality requirements*. The slide shows three *quality factors* (categories of quality requirements):

Performance. How efficient should the system be with the hardware: how fast should it respond, how many computer resources should it use, how accurate should the results be? How much data should it be able to store?

Usability. How efficient should the system be with the users: how easy to learn, how efficient in daily use, etc.

Maintainability. How easy should it be to repair defects, how easy to add new functionality, etc.

There are more quality factors than these. Each of them can be specified in many ways, as we will see later.

1.2.3 Other Deliverables
Usually the supplier delivers more than HW and SW. Documentation, for instance, will usually be specified.

The requirements can also specify services, for instance who does what in order to install the system, convert data, train users.

1.2.4 Managerial Requirements
Managerial requirements are on the borderline between requirements and contractual issues. Functional and non-functional requirements are not sufficient. You also need to know when they are met (delivery time), how to check that things are OK (validation), what happens if things go wrong (legal responsibilities, penalties), etc.

1.2.5 Helping the Reader
Large parts of the requirements specification are not requirements, but parts intended to help the reader. It could be explanation of terms (*definitions*), for instance special concepts used in the user's world. Or it could be examples, diagrams, and cross references for better understanding the requirements.
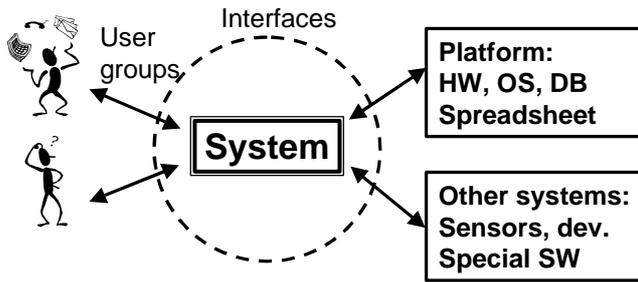
Functional requirements:
   Each interface: Input, output, sequence
   Data stored: System state
   Process: Input+state $\Rightarrow$ output+state

Non-functional reqs: **Managerial reqs:**
   Performance   **Delivery time**
   Usability   **Legal**
   Maintainability   **Penalties**
   . . .   **. . .**

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

3. Non-Functional Requirements

The non-functional requirements specify how well the system must perform its functions, for instance how fast the system responds, how easy it is to learn, how easy it is to maintain, etc. Non-functional requirements are also called *quality requirements* because they specify the quality of the system.

Requirements engineers recognize that quality requirements are very important, yet they usually pay little attention to them. The reason is that they are difficult to specify, and little help is available in literature. In this chapter we will discuss various quality requirements, particularly the different styles in which they can be specified. We will focus on requirements that can be verified.

One of the basic problems with quality requirements is that they are not mandatory in the same sense as functional requirements. A functional requirement is in most cases something the customer needs. Without it, he cannot use the system at all. Quality requirements are different. If a quality requirement is not fulfilled, the customer may still use the system - at least in most cases. So is it a requirement? In the section about efficiency requirements, we discuss ways to deal with that issue.

3.3 Usability

When the first ATMs (Automatic Teller Machines) were installed, people used them surprisingly little. Technically the ATMs worked perfectly, so why were they not used? One reason was that they were too difficult to use. Had developers been able to specify how easy the ATMs should be to use, and had they known how to develop them according to the specification, things would have been different. Over a period of many years, developers painfully improved the ATMs, and to-day most people find them easy to use.

Usability (ease of use) is still a major concern with most systems. Costly systems have failed because they had low usability, and the supplier couldn't be made responsible because the system had the specified functionality and nothing was specified about usability.

The purpose of usability requirements is to guard against that. The usability requirements must be tangible so that we are able to verify them and trace them during development. They must also be complete so that if we fulfil them, we are sure that we get the usability we intend. Meeting these goals is difficult in practice, but we can get close if we carefully select and combine the various requirements styles.

Usability factors

Before we look at the different styles, we will briefly discuss what usability is. Usability has nothing to do with program bugs or system crashes. We assume that the system works as intended by the designer. Usability is about how the user perceives and uses the system.

There are many ways to define usability, but in the following we will use this definition: Usability consists of five *usability factors*:

1. Ease of learning. How easy is the system to learn for various groups of users?
2. Task efficiency. How efficient is it for the frequent user.

3. Ease of remembering. How easy is it to remember for the casual user.
4. Understandability: How easy is it to understand what the system does.
5. Subjective satisfaction. How satisfied is the user with the system.

The different requirement styles specify and measure these factors more or less directly.

Developers often say that it is impossible to make a system that scores high on all factors. This may be true, and one purpose of the usability requirements is to specify the necessary level for each factor. As an example, a flight control system and a Web-based system for attracting new customers should have different usability factors.

Usability problems and usability testing

An important fact, confirmed by many experiments, is that *nobody can foresee the usability problems for a given user interface* - not even usability experts. Usability experts may predict many usability problems with a design, but about half of the predicted problems are false, in the sense that users don't feel they are problems. What is worse, the usability experts miss about half of the problems that real users experience.

Only some kind of testing with real users can reveal the usability problems. In order to correct the problems, we need to identify them early during development. As a result, usability specifications that cannot be tested during development have a serious weakness.

The kind of testing needed to reveal usability problems is called *usability testing*. It is an experiment where real users with appropriate background try to perform real tasks by means of the system or a prototype. Observers record the problems encountered by the user and the time to perform tasks.
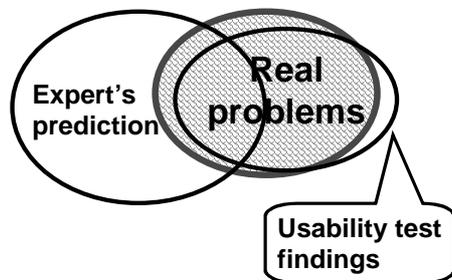
What is usability?
- **No bugs or crashes?**
- **Good on-line help?**
- **Follow MS-Windows style guide?**

Usability factors:
- **Ease of learning**
- **Task efficiency**
- **Ease of remembering**
- **Understandability**
- **Subjective satisfaction**

Finding usability problems

Expert's prediction

Real problems

Usability test findings

3.4 Usability Requirements

The slide shows six styles for usability requirements. For each style, we have indicated the risk to the customer and the supplier when using the style. The customer's risk is that although he may get what is specified, he may not get what he really needs. The supplier's risk is that he may not be able to meet the requirements - or only with excessive costs.

Performance style
R1: 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.

In the performance style we specify how fast users can learn various tasks, how fast they can perform after training, etc. We can verify these requirements through usability tests. By means of prototypes, we can make usability tests early during development, thereby tracing the requirements to design.

The style combines well with use cases. The tasks could actually be some of the use cases. Furthermore, the performance requirements might find a nice place in the use case description.

The style catches quite well the essence of usability, meaning that if the product meets the requirement, the customer gets what he needs. The main problem is to choose the right tasks

However, the style is risky to the supplier. It is not sure that the requirements can be met at all. A factor such as *efficiency in daily use* can hardly be verified during development.

Surprisingly, the style is low risk to both parties when the customer selects and buys a standard product. Why? The product is finished and the measurements can be made before the buy decision. In many cases, the supplier knows how well his product fares without having to make new measurements.

Defect style
R2: On average, a novice user shall encounter less than 0.2 serious usability defects when performing tasks Q and R. [*A serious usability problem is typically a task failure, i.e. that the user cannot complete the task on his own. Thus the requirement roughly says that at least 80% of users shall be able to complete the tasks on their own*]

The defect style resembles the performance style, but instead of measuring task times, it identifies the usability defects in the system and specifies how frequently they may occur. A usability defect is something causing the user to make mistakes or feel annoyed. The user is asked to think aloud during usability tests, and an observer records the defects.

The main advantage of the style is that the list of defects gives excellent feedback to developers, allowing them to correct the design more easily. The disadvantage is that we are less sure to catch the essence of usability. As an example, low efficiency in daily use will only be reported as a usability defect if the user complains about it.

Process style
R3: During design, a sequence of 3 prototypes shall be made. Each prototype shall be usability tested and the most important defects corrected.

The process style specifies the development procedure to be used for ensuring usability. The style doesn't say anything about the result of the development, but the customer hopes that the process will generate a good result. The example specifies iterative prototype-based development since it is recognized as an effective process, but we could specify other more risky processes such as heuristic evaluation and structured dialogue design.

How many iterations do we need? We might specify the termination criteria for the iterations, e.g. continue until no serious usability defects are left, but then we would actually have a defect style, rather than a pro??cess style. However, we could specify that more iterations shall be negotiated between customer and supplier after the three iterations. This would still be in process style.

The style is useful in many cases where developers can commit to a specific process, but not to performance or defect styles. The developer has a low risk.

The customer cannot be sure that he gets what he needs, because much is left to developers. Developers often select the wrong tasks and users for usability testing, or they only make minor changes to the prototypes rather than redesign. The result may be that usability is still unacceptable after the three iterations.

Subjective style
R4: 80% of users shall find the system easy to learn and efficient for daily use.

With the subjective style, we ask users about their opinion, typically with questionnaires using a Likert scale. This seems to catch 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. (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 outside the reach of system development. Another problem with the subjective style is that it is hard to verify the requirement during de-

	Risk	
	Cust	**Suppl**
Performance style **R1: 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.**		▨
Defect style **R2: In average, a novice user shall encounter less than 0.2 serious usability defects when performing tasks Q and R.**	▨	▩
Process style **R3: During design, a sequence of 3 prototypes shall be made. Each prototype shall be usability tested and the most important defects corrected.**	▨	
Subjective style **R4: 80% of users shall find the system easy to learn and efficient for daily use.**	▨	▩
Design style **R5: The system shall use the screen pictures shown in App. xx. (OK if design usability tested)**	▨	
Guideline style **R6: The system shall follow the MS-Windows style guide. Menus shall have at most three levels.**	▩	

velopment. Many usability experts ask users about their subjective opinion after prototype-based usability tests, but it doesn't correlate well with opinions after system deployment.

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

Design style
R5: The system shall use the screen pictures shown in App. xx. The menu points and buttons shall work as described in …

The design style prescribes the details of the user interface. Essentially it has turned the usability requirements into functional requirements. They are easy to verify in the end product and easy to trace during development. The supplier has a low risk.

Through the design, the requirements engineer has taken full responsibility for the usability. The system designer and programmer can do little to change the usability. If the requirements engineer has done a careful job with task analysis, prototyping, and usability tests, the resulting usability is adequate, and the customer runs a low risk. Otherwise his risk is high - he will not get what he needs.

Untested prototypes can be used as examples of what the user has in mind, but not as usability requirements.

Guideline style
R6: The system shall follow the MS-Windows style guide. Menus shall have at most three levels.

The guideline style prescribes the general appearance and response on the user interface. You may think of it as a set of broad functional requirements that apply to every window, etc. Guidelines may be official or de facto style guides, or they may be company guides or experience-based rules. It is possible, but cumbersome to verify and trace these requirements.

Although guidelines usually improve usability, they have little relation to the essence of usability. In other words, you can have a system that users find very hard to use although it follows the guidelines. (Such systems are actually quite common, as demonstrated by the many programs that follow the MS-Windows guidelines, yet are very difficult to use.) As a supplement to other styles, the guideline style is quite useful, particularly to help users switch between applications.

3.5 Maintenance

Most products need maintenance. Users encounter errors in them, demands grow, the environment changes. Many suppliers make quite a good business in this area, and customers are more or less forced to pay. In some cases, suppliers sell the first version of their product cheap, then collect the profit through maintenance.

To the customer the maintenance expenses can become a real burden, both because of fees to the supplier, but also because of disrupted business while maintenance goes on. The purpose of maintenance requirements is to guard against both factors.

What is Maintenance?

Usually software engineers distinguish between three types of maintenance, as illustrated on the slide:

Corrective maintenance. Correcting reported defects in the product.

Preventive maintenance. Correcting defects that haven't caused problems yet, but might do it later. A typical example is that the supplier corrects a reported defect, but ignores that it influences something else, thereby causing a problem later. Avoiding related defects is essential to both customer and supplier, since additional repairs are costly to both of them.

Perfective maintenance. Expanding the system to meet additional demands.

Maintainability requirements should deal with all these types. Another aspect is how defects are defined and what kind of repair the customer needs. Let us look at the maintenance cycle, i.e. what happens to a defect or a request for change:

Maintenance cycle

1. A problem is reported to maintenance staff.
2. The report is analyzed. Is it a malfunction or intended product behavior? is it a change request? Where is the defect that caused the malfunction, or what is the real need behind the change request.
3. The report is classified: Is it a defect or a change request? Should we reject it because it isn't really a product problem? Can we find a work-around, e.g. something the user can do to complete his task anyway? Do we have to repair it immediately or can it wait for the next release of the product?
4. Make a change of the product, either as a "fix" to be distributed immediately or as a change to be part of the next release.
5. Test the change. Also try to find and correct related defects that might turn up later.
6. Install the change and ensure that user data, setups, etc. are transferred.

Many suppliers don't have a well established maintenance cycle, and part of the requirements could be that they shall have one.
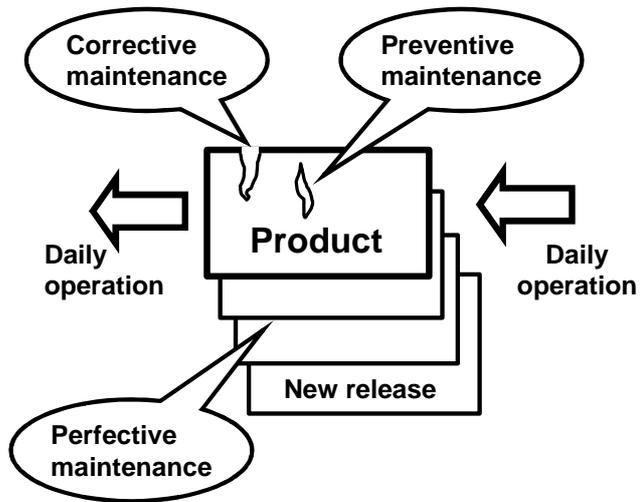
What is a defect?

Part of the cycle is to determine whether something is a defect or a change request. This can be quite difficult in practice. Typically, the contract states that the supplier has to correct defects free of charge, whereas the customer has to pay for changes.

For this reason, the supplier is tempted to claim that something is a defect only if it violates requirements. In practice it is unrealistic to specify all requirements, so defects include more than violated requirements. This point of view is acknowledged by most courts worldwide.

Some requirements are obvious, although unstated. Examples are products that eat up computer resources such as memory, so that the user has to restart the system every now and then. Another example is an accounting system that cannot handle foreign currency. Courts would consider that a defect, since such a feature is standard in to-day's accounting packages. The customer has a reasonable expectation, although unstated.

To avoid that things end up in court, the contract could contain a statement that disagreements about defects shall be decided by an arbiter nominated by both parties.

Corrective
maintenance

Preventive
maintenance

Product

Daily
operation

Daily
operation

New release

Perfective
maintenance

Maintenance cycle:

Report problem
Analyze: defect, demand?
Classify: reject? work-around?
       repair? next release?
Make change
Test
Install