Guidelines for Requirements Analysis in Students' Projects

Information Systems Group
Faculty of Electrical Engineering, Mathematics and Computer Science
University of Twente

Introduction

This document provides guidelines about how to do a requirements analysis and how to write a requirements specification. It gives hints about what you could do and warns you about things that you should not do. It is not a method that you can follow step by step. Problems are different, and what works well in one case would not be the best approach in another case. The available time can take from a few weeks to several months. And your skills and experience also determine what are good techniques to use. If a particular specification technique is treated in a course you haven't taken, then it might not be a good idea to try it out on an important job. So you have to decide for yourself what is the best to do in your project.

These guidelines are written primarily for master and bachelor students of Business Information Technology, but could be used by others. The document is self-contained, but refers to other sources for detailed descriptions of techniques. Many references are given to the book used in the bachelor course Requirements Engineering (232081), S. Lauesen: Software Requirements [Lau02]. The UT library has a copy that is permanently available (it may not leave the library).

Outline of the Guidelines

After an introductory chapter

0. What you should know before you start

the remainder or these guidelines is structured as a series of steps that comprise an idealized life cycle of a requirements specification:

1. **Analysing the problem and the problem context**
   After this step, you have an understanding of the problem context and you have learnt what should be improved and why.
2. **Defining the ideal solution**
   After this step, you know what, in principle, the best solution to the identified problem(s) would be.
3. **Defining a realistic solution**
   After this step, it has been defined what the system, for which you are going to do a requirements analysis, should achieve. Moreover, relevant stakeholders agree about its mission.
4. **Gathering requirements**
   After this step, you know what people would like the system to do and which requirements and constraints there are.
5. **Writing a requirements specification**
   After this step, you have a readable first version of the requirements specification that can be discussed with involved persons. We distinguish four separate concerns
   5.1. The contents of a requirements specification
   5.2. Specification techniques
   5.3. Readability and linguistic issues
   5.4. Quality check
6. **Validating the requirements specification**
   After this step, you have made sure that the requirements reflect what the relevant stakeholders want from this project. This is the requirements specification that you deliver.
7. **Maintaining the requirements specification**
   The world goes on, and new requirements may come up. This is outside the scope of most students' projects, but for the sake of completeness we discuss it briefly.

The ideal requirements process would follow these steps in consecutive order. As you may have guessed, the ideal requirements process does not occur in practice. But for the purpose of organising the material, it makes sense to discuss the steps one by one.

Each chapter treats a single step in the requirements specification life cycle. An outline gives essential questions that you should ask yourself (and others) and what to do about these. The remainder of the chapters treat specific topics in more detail. Appendices at the end of the document give yet more detail and references to further literature.

Not every topic is applicable in every context. Read all the outlines and study other topics as appropriate.

About this document

These Guidelines have been compiled and are maintained by the Information Systems group at the University of Twente.

Feedback is welcome! It helps us to improve future versions of the Guidelines.
Please contact Klaas Sikkel, room ZI 3102, email: k.sikkel@utwente.nl.

Contents

0 What you should know before you start 4
0.1 The requirements process 4
0.2 The requirements specification life cycle 4
0.3 From business problem to system specification 5
0.4 Why isn't there a proper method? 5
1 Analysing the problem and the problem context 7
1.1 What is the problem? 7
1.2 Organisational context 8
1.3 Stakeholders 8
1.4 Interviewing 9
2 Defining the ideal solution 10
2.1 One essential problem 10
2.2 The client's goal vs. the project goal 10
2.3 Business solution vs. software solution 10
3 Defining a realistic solution 11
3.1 Mission statement 11
4 Gathering requirements 13
4.1 Requirements at different levels 13
4.2 Modeling the system vs. modeling the system's environment 14
4.3 Types of requirements 14
4.4 Quality factors 15
4.5 Priorities 15
4.6 The Requirements Shell 15
4.7 Fit criteria 15
4.8 Requirements elicitation vs. requirements creation 16
4.9 Techniques for requirements gathering 16
4.10 Requirements elicitation for custom-tailored or COTS systems 17
5 Writing a requirements specification 18
5.1 Contents of a requirements specification 18
5.1.1 Free form or template? 18
5.2 Specification techniques 19
5.3 Readability and linguistic issues 19
5.3.1 Keep it short 19
5.3.2 Keep it simple 20
5.3.3 Structuring text 20
5.3.4 Presenting information 21
5.4 Quality check 21
5.4.1 Quality criteria for individual requirements 21
5.4.2 Consistency acrossrequirements 22
5.4.3 Have you finalized the document? 22
6 Validating a requirements spec 23
6.1 Requirements validation 23
6.2 Requirements prioritization 24
7 Maintaining the requirements specification 25
7.1 Requirements evolution 25
7.2 Traceability 25
Glossary 26
References 27
Appendix A. Context-free questions 28
Appendix B. Requirements elicitation techniques 29
Appendix C. Volere Requirements Shell 31
Appendix D. Volere Requirements Specification Template 32

0. What you should know before you start

The purpose of this chapter is to give you some general words of advice. You should read this before you start your requirements analysis.

Way of thinking – What are the essential questions?

- What is a requirements specification?
- How do you obtain a requirements specification?

0.1 The requirements process

Requirements analysis is for a large part a social activity. The requirements analyst's job is to find what relevant stakeholders want and lay that down in a suitable specification (and not to invent the requirements himself). Gause and Weinberg [GW89] define a *requirements process* as

> the part of [system] development in which *people attempt to discover what is desired*.

In the early days of computing, it was thought that the requirements analyst's job is to find out what is *needed*. This presupposes that there is some objective need, and analysis will reveal what that need is. In many projects, this is not the case. There are various things that could be desired for various reasons. Moreover, many relevant persons do not have a clear picture of their own desires – the process of requirements discovery helps them to find out what they really want.

To make things more complicated, any project has a number of different stakeholders with different interests, and it is usually not feasible to incorporate all desires of all stakeholders. Choices have to be made and somebody has to put some effort into making the stakeholders accept the resulting requirements specification.

0.2 The requirements specification life cycle

In this section we elaborate a requirements specification life cycle of seven steps. In the next section we will argue that it doesn't work that way, and in practice you won't be able to strictly separate these steps.

What, then, is the point of introducing this model? It's a *reference model*, describing the ideal case. Even though you will never meet the ideal case, it helps to keep structure and put things in the right place. For example, if you return from a chaotic focus group meeting which has done bits of steps 1, 2, 4, and 6 in random order, you can get some structure in your equally chaotic notes by ordering them according to these steps.

It's like the waterfall model in Software Engineering – the first thing you learn in an SE course, despite the fact that nobody ever could make it work that way. It's the lucid enumeration of steps that makes it worth knowing it.

In the generic requirements process described here we distinguish different phases

- Finding out what the problem is, and what kind of solution is desired (steps 1–3)
- Drawing up a requirements specification for the desired solution (steps 4–6)
- Maintaining the requirements specification when requirements change later on in the project (step 7)

In each phase we can distinguish four different kinds of activities:

- *Preparation*: getting organized before you start, finding out what you are going to do and whom you may want to talk to, etc.
- *Elicitation*: going out and finding requirements, by asking people, observing, reading documents, etc.
- *Engineering*: putting things together: specifying what elicited and observed, organizing and combining things. There is always an element of *design* involved.
- Negotiation and decision making. This is *politics*, rather than engineering, but is an inevitable part of getting a requirements specification accepted.

The complete life cycle model is shown in Figure 1. The phases cycle through the different activities, yielding our seven steps:

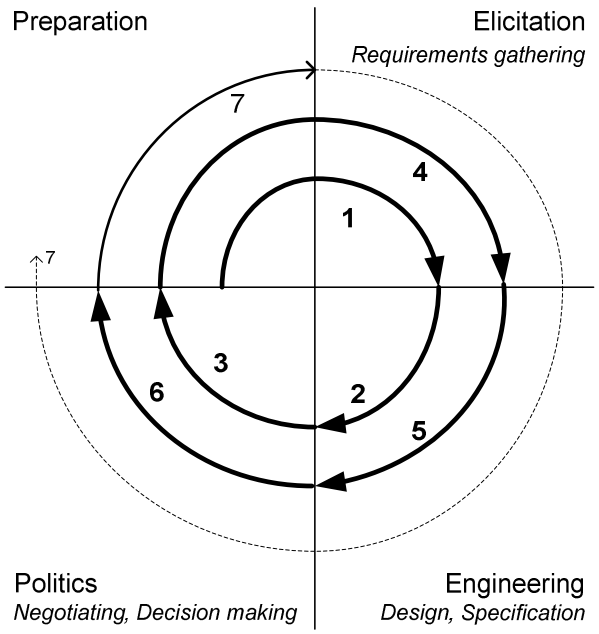

Figure 1: The requirements life cycle

1. Analysing the problem and the problem context
2. Defining the ideal solution
3. Defining a realistic solution
4. Gathering requirements
5. Writing a requirements specification
6. Validating the requirements specification
7. Maintaining the requirements specification

The maintenance phase is never finished and can cycle on forever. (But we can anticipate this).

0.3 From business problem to system specification

Another way to look at the relation between problem and solution is shown in Figure 2.

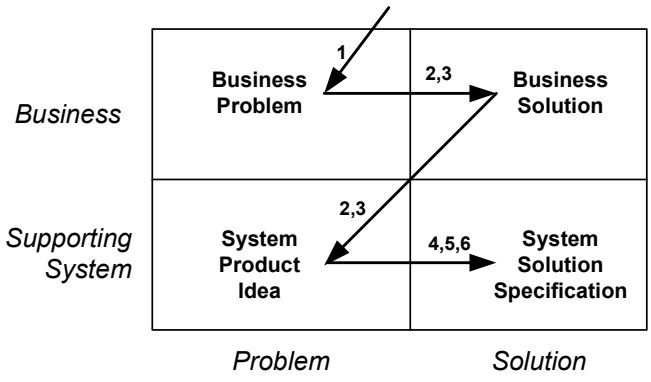

Figure 2: The Z model

We distinguish between problem and solution, and between business and supporting (software) system. In a perfectly rational world, a requirements analysis process would follow the arrows in the diagram.

In a narrow sense, requirements analysis is only concerned with the last arrow. Somebody has suggested that a system for a particular purpose can be developed (or bought) and your task as a requirements analyst is to find the requirements for that system. However, in order to find these requirements, it is important to know *why* this system is needed, what problem it will solve – otherwise it's not possible to determine the requirements.

A problem *always* arises in the real world. Even when it's clear that the system is to blame. E.g. "our system is too slow." It would not be a problem if people would not depend on that system for doing the particular job they do. In other circumstances (e.g. the same company 5 years ago) the same system might not be experienced as being to slow. The idea to design, replace, or upgrade a system doesn't arise because having the system is a goal in itself, the system is needed for some purpose.

It is called "business problem" because most requirements engineering is done for systems that have some business purpose, but it doesn't have to be related to commercial business.

The solution to a business problem is always a business solution. It is possible that this solution involves a computer system. It is tempting to think that acquiring a new system may solve a business problem (this is a mistake that is often made). *Using* a new system can be the solution to a problem. Acquiring the system isn't suffficient, the system has to fit into the way the work is done – or perhaps the work has to be reorganised, so as to exploit the capabilities of the new system.

In perfectly rational top-down design process, one would first define a business solution to address the business problem, then consider what kind of system is needed to support the business solution and finally draw up a requirements specification. After the arrows in Figure 2, this is called the Z model.

To make sure that we do requirements analysis for a system that helps addressing the *right* problem, we start with step 1 – identifying the problem. Steps 2 and 3 yield an idea of the solution and the system needed to realize that solution. After that we can do a more detailed requirements analysis in steps 4–6.

At least, that's the theory...

0.4 Why isn't there a proper method?

Life would be a lot easier with a method that you could follow step by step. Unfortunately, our life cycle model doesn't pretend to be that kind of method. In fact no such method exists for requirements analysis.

There is no method that addresses all cases

For each project you have to decide which issues are important and need a lot of care, and which issues are trivial or do not apply. These guidelines are no substitute for thinking for yourself, and *you* have to judge what is needed in your project.

Requirements analysis projects differ a lot in scope and nature. Some examples from projects carried out by M.Sc. students:

1. A commercial bank has a problem with customer loyalty. Obtaining new customers by means of marketing actions seems to work, but the bank isn't able to retain these customers for a long time. Can appropriate CRM software help them to increase the loyalty of their customer base?

The focus in this project is more on organizational practices than on the technical support system. In this project something was implemented in the end, but initially it was not at all clear what the solution should look like. But it was evident that a system won't help if the bank's employees are unable or unwilling to use it properly. Steps 1–6 were carried out, but the emphasis was on steps 1, 4, and 6.

2. A telecom company wants to find out how it could rent telephone services to corporate clients, making use of VoIP (Voice over IP) technology.

This is primarily a technical project. Not much study has to be done about how people would use a VoIP telephone, because it should work as a regular telephone, and possibly clients shouldn't even be aware of the difference. Steps 3–6 were carried out in this case (the result of step 2, the ideal solution, was given as a starting point for the project) but the emphasis was on steps 4 and 5.

3. The Police department in a region in the north of the Netherlands has difficulties in providing statistical material to the Ministry of Justice. Sometimes when the Ministry asks for statistics about a particular type of crime, they have to go through all the database records to find the requested numbers by hand.

The stated problem is clear, but it is a symptom of an underlying problem that was hard to find and harder to solve. In this project only steps 1–3 were carried out.

The steps in a requirements analysis process do not take place in consecutive order

Only in the ideal situation, you do step 1 first, then step 2, and so on, without retracing your steps. In practice you will find it hard to separate analysing the problem (step 1) from eliciting the requirements (step 4). Also, it makes sense to combine requirements elicitation (step 4) with writing down the elicited requirements (step 5).

Many projects, and some excellent requirements analysis methods, start with step 3. If the project goals are straightforward and you are asked to draw up a requirements specification for a system with a clear purpose, step 3 is a natural starting point. This implies that *somebody else* has already performed steps 1 and 2, found out what the problem and the ideal solution was, decided to set up a project and engage you as a requirements engineer. If this is the case, you can – and should – find the results of the problem analysis. If these don't exist, e.g. if the project is driven by a solution, rather than a problem, you should consider doing some problem analysis after all.

However, in many cases, including most cases in which our students do a requirements analysis, there is some idea about the problem, but it is not immediately obvious what the best solution is – otherwise they wouldn't have asked the university.

Many systems fail, despite the fact that they fulfil the requirements, because the problem is poorly understood and a solution is built that doesn't address the real problem. For this reason we insist that step 1 is part of the requirements analysis.

Problem-solution co-refinement

It's a very good idea to define the problem first, and then the solution. If it's a difficult problem with no easy solution, there is a complex relationshop between problem and solution. The nature of a possible solution determines what problems you can solve, and if we don't know the solution yet we might not know exactly which problem we *can* solve. Empirical studies have shown that refining the solution and refining the problem go hand in hand [Cro89]. That's why you always have to do some rework on previous steps, no matter which method you follow.

The method does not work

You do the work. The method is just a set of guidelines. The method is not responsible for your work products, nor are the authors of the method. You are responsible yourself.

Step 1. Analysing the problem and the problem context

The purpose of this step is to find out what the problem is and, equally important, to understand the situation in which the problem occurs. It is not the purpose of this step to think about possible solutions. That comes later, after we have learnt enough about the problem.

Way of thinking – What are the essential questions?

- What are the problems (goals, desires) and what are the causes for these problems?
- Is the stated problem the real problem or it is a symptom of an underlying problem?
- Who are the stakeholders?
- What will be the impact if the problems are resolved / the goals are accomplished?

Approach – How to find answers to these questions?

It makes sense to learn something about what is going on, what are the causes for the problems and which parties have an interest in (not) solving the problem. To that end you have to do two things:

- identify (groups of) stakeholders
- interview relevant persons

Your supervisor or the client can help you drawing up an initial list of persons you might want to speak to (and talking to these you may become aware of other stakeholders to be considered). If there are relevant documents about the current system, it could be worthwhile to read those first. If you know what you're talking about, you'll get better results.

The list of "context-free questions" in Appendix A could be a good starting point. Some other points are elaborated below.

Product – What do you write down?

Lay down your problem analysis in a short paper. Target audience for this paper are the stakeholders. They should be able to find out, as easily as possible, whether you have captured their problem appropriately. Hence it is important that the analysis is easily readable and to the point. Making it short and readable is a *lot* more work than just summing up what you've found. But it's well worth the effort if you want to get feedback and gain credibility with the client and other stakeholders.

Follow-up – What do you do with this document?

- Make sure that you have a good enough version (if possible, consult your supervisors)
- Circulate it to relevant persons and ask for their feedback
- If needed: adapt it, based on the feedback
- Include the adapted version as a chapter or an appendix to your final report.

1.1 What is the problem?

How much time, effort and skill it takes to identify the problem varies from case to case.

There are (few) projects in which the problem is clear. Consider a project to develop a prototype for some technologically innovative gadget. You may find it interesting to know what people eventually will do with it, but the prime challenge in *this* project is in getting the technology working.

In some projects, finding the problem is very hard. For example in a situation where key persons have hidden agendas, it needs skill and tact to find out what is going on.

In some projects, the problem *appears* to be clear. But the problem that people experience is a symptom of a deeper, underlying problem, and it makes a lot more sense to solve the real problem than to address the symptom.

Problems at which level?

If you ask people which problems they experience, they often will tell you that properties of the current system (or their absence) are a problem. This is experienced as a problem, it directly bothers people. The *real* problem, however, is that they cannot perform some task effectively or efficiently. Adapting the system functions they complain about can be, but need not be the best solution. Perhaps

is it better to reorganize the work, or to replace the whole system rather than to repair some functions.

Do not just ask what the problems are, always ask *why* this is experienced as a problem. Sometimes you have to ask "why" several times to find the real reason behind the reason behind the reason behind the problem.

Problem vs. solution

When you ask for problems, many people (including most students not trained in requirements engineering) will come up with solutions.

- A problem is a difference between what is experienced and what is desired.
- A solution is a way to reduce a problem

These two are related, but different. It is possible that there are different solutions for the same problem.

If you inquire about problems you may be told, e.g. "we need an ERP system." What is stated here is the absence of a solution. Again, we need to go up one level, and ask "why". There could be various reasons. Perhaps implementing an ERP system is indeed the best solution, perhaps there are also other solutions worth considering.

How important is a problem?

Not all problems are equally important. One way to get an indication is to ask the following questions (costs and benefits are not only financial).

- What are the costs when this problem is solved?
- What are the benefits if this problem is solved?
- What are the costs if the problem is not solved?
- What are the benefits if the problem is not solved?

If you want to get an idea about the *urgency* of a problem, you could add

- What are the costs if the problem is solved after one year?
- What are the benefits if the problem is solved after one year?

More about problem analysis

A course Problem Analysis and Software Requirements (232080) is part of the BIT master programme.

1.2 Organisational context

How is the project positioned in the organisation?

- How does the project fit in the organisation's strategy?
- What does management think about this project?
- Who is responsible for the project's funding (the client) and who is responsible for managing the project?

Goals

A problem is a problem because it prevents some goal from being realized. In perfectly logical world, you would first write down the goals and then look for problems obstructing these goals. Eliciting goals is a lot more difficult than making a list of problems. Many people are not willing or able to state their goals. Try to get some idea about the following issues:

- What are the goals of the organisation?
- Which personal goals (which are usually hidden) also play a role?
- What are the goals of the organisational unit? Are these different from the goals of the organisation as a whole?

The official goals of the organisation (typically: running the primary process effectively and efficiently) give some hold, and can be used in your problem analysis to motivate why a solution is needed. But keep an open mind for what is going on around you.

1.3 Stakeholders

A stakeholder to a project is someone who gains or loses something (could be functionality, revenue, status, compliance with rules, and so on) as a result of that project [AR04].

Stakeholders include

- the client (who pays for the system development),
- customers,
- system developers,
- direct users (who will work with the system),
- indirect users (e.g. who will get information from the system),
- system operators.

And there could be others, e.g.

- government bodies, having an interest that the law is not violated.

Alexander [Ale03] gives a simple but powerful model of stakeholder roles that can help you discover the stakeholders for your project.

In some cases you may consider an organisation or company to be a stakeholder. It is always better to think of concrete persons, rather than abstract bodies. ("Mr. Smith in the procurement department", rather than "company A&B"). A stakeholder group is *homogeneous* if all persons in that group want the same thing. This is not always the case.

If you want to involve stakeholders in the requirements process, you have to determine who represents a stakeholder group. There are several forms of representation:

- exhaustive (everybody in the group)
- representation by sample (choose the sample carefully of the group is not homogeneous)
- representation by surrogate (somebody who knows a group of stakeholders quite well).

Representation by surrogate ("our marketing department knows what our customers want") is always risky. If you don't have access to real users, you must read *The Inmates are Running the Asylum* [Coo99] before you attempt to write down other people's estimate of what the users would desire.

Stakeholders have different problems. Even in the unlikely event that there is only a single problem, stakeholders will experience this problem differently.

If you want to get clear which stakeholder has which problem, you could make a schema as follows:

Stakeholder ***Problem***	A	B	...
Problem 1			
Problem 2			
...			

1.4 Interviewing[1]

Interviewing is the most often used technique to learn about problems. It works fine, if you are aware of its limitations.

When you ask people about their daily tasks, they have difficulties explaining what they do and why they do things the way they do. Some people have hidden agendas and will not give honest answers.

Make sure you have the right interview partner, and not a surrogate. If you want to know the problems on the shop floor, you should talk to the people who do the work there, not to their managers.

Prepare yourself for the interview. If you know what you're talking about you will get a better response. Make a list of questions. The context-free questions in Appendix A can serve as inspiration. If you can make these questions more specific for the situation, that's better.

Despite this, an interview is not a question-and-answer session. Start with one issue, and most likely the interviewees will cover a number of questions when you let them talk. If they bring up issues that are relevant, but not on your list, even better. Use your list to check whether the issues are covered. If something hasn't been touched upon, you may bring it up.

When you discuss day-to-day problems with an unsatisfactory system, ask about *critical tasks*. When does the user work under stress? When is it important that nothing goes wrong?

As a general rule you should be polite and sensitive to the interview partner. Some people don't like to admit that they have problems. There is whole range of euphemism that roughly mean the same thing: challenges, things you find hard to deal with, concerns, issues, things that could be improved, ...

Some managers get offended if you ask "why", as they are not used to be questioned about their motives. If asking "why do you do this" doesn't work, you may ask "when do you do this" as a substitute.

[1] largely based on [Lau02], section 8.8.2.

Step 2. Defining the ideal solution

Armed with sufficient knowledge of what the problems are, we can start to think about a solution. Usually it makes sense to do that in two steps. A realistic goal – the subject of step 3 – is constrained by practical limitations. The purpose of this step is to find out what the client would *like* to achieve.

Way of thinking – What are the essential questions?

- What is the essential problem?
- What would be an ideal solution to this problem?

Approach – How to find answers to these questions?

If there is a single problem and everybody agrees that this is the problem that needs to be solved, step 2 is easy. If, however, there are various issues and different stakeholders experience different problems, this is not trivial. It has to be decided, somehow, what the essential problem is. In that case you have to discuss it with the client or perhaps organise a focus group with different stakeholders (see 4.5).

Product – What do you write down?

A brief text (maximum one page, preferably half a page) describing
- the essential problem,
- the proposed solution,
- a brief explanation about the motivation of the essential problem and the choices you made.

If there was a group session, you probably have a list of other problems and possible solutions. The explanation should make clear why *this* problem was chosen as the essential problem.

Follow-up – What do you do with this document?

This is not an official document (achieving the ideal solution is not an objective of the project), but it could be the most important page in the whole project. Check informally whether relevant stakeholders can agree with it. If they can, there is agreement about the focus of the project.
If, on the other hand, it turns out that some stakeholders have serious troubles with the choice of the essential problem or solution, you have achieved your first success! You have shown that the matter is more complicated and delicate than the client thought, and identified a potentially fatal risk for the project.

2.1 One essential problem

The goal of the project is to solve, in the best possible way, the essential problem. The solution may partially solve other problems as well, but the priorities must be clear. If you have multiple goals, all equally important, then sooner or later you will face design decisions that cannot fully satisfy these goals simultaneously and you'll have to favour one goal at the expense of another.

2.2 The client's goal vs. the project goal

There is a difference between *the external goal* or *client's goal* (what the client wants to achieve, e.g. increased sales) and the *project goal* (what the project intends to deliver, e.g. a system to support the sales process). The external goal provides a motivation for the project goal.

2.3 Business solution vs. software solution

The external goal is always to find a solution to a business problem (see the Z model in 0.3). The project goal could be on the software level (otherwise you weren't asked for a requirements analysis).

If the project goal is to come up with a software solution specfication, you should spend some words on the business solution to which your software solution will contribute.

Step 3. Defining a realistic solution

The purpose of this step is to define a realistic solution and to gain acceptance for it.

Way of thinking – What are the essential questions?

- What is a realistic solution?
- What needs to be done to get support for this solution?
- How can the migration to an improved situation be accomplished?

Approach – How to find answers to these questions?

There could be all kinds of reasons why the ideal solution is not achievable. Budget limitations are a mundane but common example.

It is not always clear whether a solution is acceptable for various parties. If an important stakeholder strongly objects to the solution, it is not a good solution (even though you may find his reasons irrelevant). Acceptance can be increased by involving the right persons in the right way.

If difficult choices have to be made, they are for the client, not for you to make. But you can support the client in making the right choice by providing clear alternatives with their consequences.

Issues to think about:
- Which factors determine the success of the project?
- Which resources are available for the project?
- What is the attitude (motivation, acceptance) of the intended users?
- Which resources (funds, courses, etc.,) are available for migration?

Product – What do you write down?

Write a realistic mission statement. Desired properties that will not be realized are to be listed as exclusions.

If you think there could be problems with the migration to a new solution, it makes sense to make an outline of a migration plan.

Follow-up – What do you do with this document?

The mission statement is a formal document, to be incorporated in the requirements specification. Show it to all stakeholders (which can lead to minor changes) and make sure that it is approved by the client.

3.1 Mission statement

There are various definitions of a mission statement. Wieringa [Wie03, Ch. 5] describes the mission statement according to Yourdon. We use a slightly different format; the suggested solution need not be limited to a computer system. The system can contain people and procedures, and need not even involve a computer system.

A mission statement describes the following points

- A short motivation
- System boundary (is it a computer system, or a system that includes people around the hardware/software)
- The goal of the system (which problem will be solved)
- Exclusions (which problems will not be solved)
- How the problem will be solved

An explanation can be added as to why certain issues are (not) treated. This explanation is not part of the mission statement proper.

If different stakeholders have different interests, you could formulate alternative mission statements, and ask the client to make a choice. As stated in 2.1 a project should pursue one prime goal. Having a mission statement that is a compromise between different goals is asking for trouble later in the project.

The final version of the mission statement should be known, understood, and accepted by all important stakeholders. That doesn't mean that stakeholders agree about what they desire and what would be

ideal. It means that they agree that this is the mission for this project.

Example of a mission statement

The following mission statement is taken from a recent M.Sc. project. It has five paragraphs which could be labelled: introduction / type of system / goal / exclusions / solution. The external goal is given in the first paragraph as a motivation for the project goal in the third paragraph. The system boundary is not stated explicitly, evidently(?) it is a software system.

The purpose of each paragraph is clear, so there is no need to include headers.

> *A problem to be solved in electronic commerce is the specification of terms of delivery in such a way that can it can be established beyond doubt – if necessary, in court – what these terms were at the time the contract was made. The E-Terms consortium wishes to address this problem by establishing an E-Terms repository. When a business party submits terms to the repository, the consortium guarantees that the applicable terms can be retrieved unaltered by any interested party at any future moment.*
>
> *In this project [student] will develop a prototype of an E-Terms repository.*
>
> *The purpose of the prototype is to serve as a proof of concept, aimed at showing the possibility of creating a repository and functioning as a guide for the development towards a final version. Furthermore, the prototype will be used in the external promotion of the concept to potential users, submitters and developers. It should serve both to increase the interest in the E-Terms service and to gather relevant feedback from interested parties.*
>
> *Efficiency and reliability requirements envisaged for the final product need not be met by the prototype repository.*
>
> *[Some words about the different functions to be supported by the E-terms die door de repository.]*

Step 4. Gathering requirements

The purpose of this step is to find out what people would desire the system to do, which demands they have, and which constraints there are.

Way of thinking – What are the essential questions?

- Which kind of requirements are needed?
- How and where can I find these requirements?
- Which questions do I ask?
- Could I have missed any important requirements?

Approach – How to find answers to these questions?

A common way to find requirements is to interview people. If you did that in step 1, you may already have collected some requirements. With a clear project goal and mission, it could happen that you want more specific requirements from persons you talked to earlier.

A number of other techniques are listed below. Obviously, it depends on the context and the kind of system which technique is most suitable, and which stakeholders to involve.

We make a distinction between *business-level requirements* and *system-level requirements* (elaborated below in Section 4.1) System-level requirements describe what the system should do. Business-level requirements describe which tasks should be supported by the system. Traditional software engineering has a focus on system-level requirements. However, if the main challenge is to find out how the efficiency of a task or an organisation can be improved, it could be worthwhile to focus on the business-level requirements.

Product – What do you write down?

You have written notes of all the requirements you gathered and other relevant information that people gave you.

Follow-up – What do you do with this document?

Writing an easily readable requirements specification, based on your notes, is still a lot of work. That will be the subject of step 5.

4.1 Requirements at different levels

Consider an information system for the reception desk at a hotel. It could have the following requirements:

R1. *The system shall allow the hotel to increase its bookings with 15 % without adding reception staff.*

R2. *The system will support the receptionist to prepare for the arrival of a tourist bus.*

R3. *The system shall be able to record that a room is occupied for repair in a specified period.*

R4. *The system shall record the data specified in the Class diagram in appendix X.*

We can make a distinction between business and system and between problem and solution, as illustrated in Figure 3. The requirements R1–R4 describe a business goal, business process, system requirement and system design, respectively.

	Problem	Solution
Business	**Goal-level requirements** *business goal*	**Business-level requirements** *business process*
Supporting System	**System-level requirements** *system requirements*	**Design-level requirements** *system design*

Figure 3 – requirements levels[2]

[2] Astute readers will have noticed a difference between figures 2 and 3. In the Z model in Figure 2, it was suggested that the requirements specification, produced in steps 4, 5, 6, provides a *solution* (bottom left corner). In Figure 3, system requirements are stated as a *problem* (bottom left corner). This paradox is caused by a

Most relevant are the business process and system requirements – assuming that the focus of your requirements specification is to make clear how a proposed system can support an envisaged business process. But we discuss each of them and give them a name for easy reference.

- **Goal-level requirements** describe a business problem, i.e., a goal that the client intends to achieve. This is an external goal (see 2.2); the supplier of the system can never guarantee that goal will be achieved, hence it is not a project goal. It could be useful to know the business goals of the client (you want the client to be happy with the delivered system), but goal-level requirements are not usually part of a requirement specification.
- **Business-level requirements**[3] describe business process: they deal with tasks to be supported by the system – without being specific about which system functions are needed to do so. The normal check-in procedure in a hotel has been designed for guests who come alone or in small groups. If a bus with several dozens of guests arrives, the reception will follow a different procedure in which the administration is done in advance, perhaps printing a list of guest names and room numbers. Which particular solution is to be chosen isn't important at this stage. The requirement in this example is that the system allows the staff to handle the exceptional situation in an appropriate manner.
- **System-level requirements**[4] specify a software problem, i.e. the desired behaviour of the system: individual functions of the system (functional requirements) and overall quality properties of the system (quality requirements).
- **Design-level requirements** specify a software solution, i.e., details about how a particular function of the system is to be implemented. These should be used sparingly in a requirements specification, it is not meant to give a detailed design of the system. But sometimes problem and solution are hard to separate. A class diagram is a good example: by specifying the object classes and their relations, it becomes clearer which information can be stored in and retrieved from the system.

In Software Engineering, the focus is on technologically challenging projects, rather than embedding the technology in an organizational context. In that tradition, software requirements are system-level requirements. In Software Engineering handbooks, finding business-level requirements is done in a separate, first phase of the software life cycle, which they call system analysis or information analysis.

In Information Systems, the biggest challenge in a project is often to make sure that a system fits the context in which it is to be deployed, rather than the technical development of the system itself. Therefore we have a broader view of requirements analysis and explicitly include the business level.

System-level requirements tell us *what* the desired properties of a system are. Business-level requirements tell us *why* a system must have certain properties.

4.2 Modeling the system vs. modeling the system's environment

Typically business-level requirements are about the system's environment, and system-level requirements about the system itself. But the system environment is not limited to the business level. Systems usually have to exchange data with other systems, which may cause requirements at the system level and even at the design level.

A requirements specification should contain a model of the environment, including other systems it has to interface with. A context diagram (see, e.g., Lauesen [Lau02, section 3.2], Wieringa [Wie03]) is a good high-level description of a system's environment.

4.3 Types of requirements

Requirements come in different types. In a requirements specification you may find the following categories:

- **Constraints.** These are global requirements that restrict the way you produce the product. Budget and delivery deadline are constraints. There can also be technical constraints, e.g. that the system should run on particular hardware or interface with an existing legacy system. Usually you are not at liberty to negotiate changes to constraints.
- **Data requirements.** A requirements specification could have a data model, specifying the kind of data that have to be stored in the system, e.g. in the form of a UML class diagram.
- **Functional requirements.** These describe the functions of the system. This can be on the system level or on the business level. In the latter case, functional requirements describe the tasks to be supported by the system.

---

difference in level of abstraction. Figure 2 takes the perspective of the first iteration in the requirements life cycle, steps 1, 2, 3. Defining how the system will behave is, at that stage, a solution to the real-world problem that needs to be solved. Figure 3 is takes the perspective of later iterations of the life cycle: the requirements are regarded as a problem statement, the solution is realizing a system that meets these requirements. Problem and solution are not absolute categories: some person's solution is another person's problem. A solution at a higher level is a problem at a lower level.

[3] Lauesen [Lau02] calls these "domain-level requirements," another term often found in the literature is "user requirements."

[4] Lauesen [Lau02] calls these "product-level requirements."

- **Quality requirements,** also called **non-functional requirements.** These describe quality properties of the system as a whole, see 4.4 below. Not all properties are relevant for each system.

Many examples of these types of requirements are given by Lauesen [Lau02].

4.4 Quality factors

Different sources give different classications for quality factors, but they usually overlap. ISO 9126 distinguishes

- Functionality (accuracy, security, interoperability, suitability, compliance)
- Reliability (maturity, fault tolerance, recoverability)
- Usability
- Efficiency
- Maintainability (testability, changeability, analyzability, stability)
- Portability (adaptability, installability, conformance, replaceability)

For large, safety-critical systems there could be requirments for all the second-level quality factors mentioned in parentheses. Probably you need to address only the main categories.

Usually there are trade-offs between quality factors. Increasing the security may decrease the usability of the system, and reversed.

In the initial stages of requirements elicitation, it is very difficult to get measurable quality requirements. What you really want to know, initially, is the relative importance of various quality factors for the project you're working for. Is security a really big issue, or is it only marginally relevant? If the system would be down for half a day, what would be the consequences for the customer?

For quality factors that really matter, you should try, later on, to get measurable requirements – see 4.7: fit criteria – otherwise there is no way of knowing whether the system, when it is delivered, meets the requirements.

4.5 Priorities

In the process of requirements gathering, you want to get an idea how important the various requirements are. It is possible that not all the demands and desires can be fulfilled, so it useful to know what could eventually be dropped. At a later stage (step 6), when there is a complete list of requirements, priorities can be ranked and negotiated, if necessary. At this stage, you want a first indication.

MoSCoW

For a rough indication you can use the so-called MoSCoW classification:

- **Must**: essential requirements, the system must meet these
- **Should**: requirements that the system should meet, if possible
- **Could**: nice features, that could be included if it doesn't take too much time and effort
- **Won't**: exclusions, i.e., features that some stakeholders would consider reasonable requirements, but, for some reason or other, will not be included in the system

Customer satisfaction and dissatisfaction

The Volere method [RR99] suggests estimating, on a scale of 1 to 5, customer satisfaction and dissatisfaction.

- **Customer satisfaction** is a measure of how happy the client will be if you successfully deliver an implementation of the requirement.
- **Customer dissatisfaction** is a measure of how unhappy the client will be if you do not successfully deliver this requirement.

Customer satisfaction and dissatisfaction need not be each other's inverse. For example: a very nice feature in the "could" category could make the client really happy (satisfaction = 5), but, since it's not necessary for solving the essential problem, he is not going to be deeply disappointed if it doesn't materialize (dissatisfaction = 3). Another example: If a system is supposed to be online 24/7, availability is taken for granted (satisfaction = 3), but poor availability is problematic (dissatisfaction 5).

It is generally a good idea to ask customers for (dis)satisfaction rates.

4.6 The Requirements Shell

In the Volere method [RR99], Suzanne and James Robertson give a template to be filled in for each requirement. They call it the Requirements Shell. It is suggested that you carry cardboard copies of the template with you when go around gathering requirements. See Appendix C.

4.7 Fit criteria

The Volere Requirements Shell template makes a distinction between the *description* of a requirement (what you want) and the *fit criterion* (how to determine whether what you want has been achieved). A requirement with a fit criterion is measurable: there is a way to determine objectively whether the requirement is satisfied by a given product.

For data and functional requirements this is not too difficult; if the requirement is complete and unambiguous there is no room for discussion whether a particular solution does or does not satisfy the requirement.

Quality requirements are usually harder in this respect. You may have gathered some requirements that have a description but as yet no fit criterion. E.g.

The system must respond [...] fast.

This is a clear desire, but not measurable. A fit criterion should tell precisely how fast.

The system must respond [...] within 2 seconds

is clear enough. However, is it necessary to guarantee that *all* responses are within 2 seconds and, say, 2.2 seconds during peak load is not acceptable, even if this would greatly increase the cost of the system?

A typical form for such a requirement is

The system must respond [...] within 2 seconds in 90 % of the cases and always within 5 seconds.

This is a usual form for such requirements and the fit criterion is okay. Yet, you could ask yourself wether these values are arbitrary (in which case other values can be negotiated if these would cause problems) or derived from some specific purpose. *Rationale* is another slot in the requirements shell. If you get to know why response time is an issue, but the proper values cannot be estimated right now, you should at least capture the rationale, e.g.

The system must respond [...] not slower than comparable systems.

This has no proper fit criterion yet, because it isn't defined what comparable systems are, but for the time being it expresses approprately what is desired.

Another possibility is to give a template for a fit criterion and leave it to the system provider/designer to suggest a reasonable value:

The system must respond [...] within __ seconds in __% of the cases and always within __ seconds.

For a response time requirement we know at least that *time* is the dimension in which a fit criterion has to be specified. For some other quality requirements there is not even an obvious choice for the dimension in which quality can be measured.

Usability

Usability is one of the hardest things to quantify. Lauesen [Lau02, Chapter 6.7] gives 9 different ways to specify measurable usability requirements. Some examples:

U1. *Novice users shall perform tasks Q and R in 15 minutes. Experienced users complete tasks Q, R, and S in 2 minutes.*

U2. *80 % of the users shall find the system easy to learn. 60 % shall recomment it to others.*

U3. *Three prototype versions shall be made and usability tested during design.*

4.8 Requirements elicitation vs. requirements creation

Finding requirements is traditionally called "elicitation", which means "uncovering". Implicitly it is assumed that there are some objective needs, and it is the task of the requirements engineer to find out what those needs are. Gause and Weinberg [GW89] made clear that in most cases requirements are not elicited but *created*. The customer usually hasn't thought about the details, and the requirements analysis process may help him to explore possibilities and/or force him to decide what he wants.

In requirements *elicitation,* you are like a scientist studying the behaviour of planets: you observe what happens but you do not influence it. Requirements elicitation is simply writing down the requirements as they are told to you by stakeholders. In requirements *creation,* on the other hand, you work with the customer to identify the requirements. You join the customer in the search for goals to achieve and problems to solve. In the first case, the customer knows what the requirements are and you help him or her to write these down. In the second case, the customer does not know what the requirements are and you work with him or her to determine what they are. Requirements elicitation in its pure form does not exist.

4.9 Techniques for requirements gathering

Common techniques include (but are not limited to) the following. Lauesen [Lau02] gives some more details. Appendix B gives a longer list with further references.

- **Interviewing.** (See step 1)
- **Documents.** If the purpose of a project is to replace an existing system, the documentation of that system can give useful information, e.g. data models. Also, if you studied documents in step 1, for finding the goals and background of the project, these may hint to requirements. It is always useful to cross-check what you read in documents with what you hear in interviews.
  In an IT-intensive organisation there could be architecture documents with guidelines and constraints for individual applications.
- **Observation.** The way people work is not necessarily the same as the way people *think* they work or the way they *describe* how they work. To be a good observer, you need some skills (not taught in our courses). See Beyer & Holtzblatt [BH98].
- **Brainstorming.** You should have experience with brainstorms if you want to moderate one.
- **Focus groups.** In a focus group, representatives of different stakeholder groups come together to identify problems, needs and possible solutions. Lauesen [Lau02, section 8.4] describes how to organize focus groups.
  If you get people to attend a focus group, they are motivated to discuss problems, requirements, and solutions, and you should allow for that. You cannot limit a focus group to a single step of our life cycle, but you can emphasise one step of our

life cycle. There is always some overlap with other steps.

- **Prototyping.** Prototypes can help to imagine what the system could be like and thus to be more concrete about what they (don't) want. A prototype is typically a mock-up, in which the functionality is faked or absent. For a very first impression, a sketch on paper will do as well.
- **Study similar companies.**

4.10 Requirements elicitation for custom-tailored or COTS systems

Most requirements analysis methods deal with the case that a new system has to be developed, for which requirements need to be drawn up. In many cases, however, there is no need to develop a new system – you can buy one. Software that you can readily buy is called common off-the-shelf (COTS) software.

When a COTS solution is sought, some steps in the requirements process differ from our general outline.

Another possibility is that a commercial system is bought, but more work (fine-tuning, interfacing with other systems) needs to be done in the operation environment. This is typically the case with ERP systems. If there is a choice of different suppliers, this would call for a tender process.

After the project goals and mission are clear, some alternatives to are

- **Tender process**. You draw up a requirements specification of what is needed, and ask different vendors whether they can supply this, and at what price.
- **COTS selection.** If different companies sell software packages with the same kind of service, you have to select which one is the most suitable. Chances are that the functionality of these packages is rather similar (if they wouldn't satisfy the *market requirements*, i.e. the functions that such a package ought to have, they wouldn't be in business). There is usually more difference in quality issues (e.g. how good is their service?). Hence the selection should pay due attention to these.

If your client is a vendor of COTS software, some of the items in these guidelines have to be reinterpreted accordingly. It is important as ever that the product satisfies the customer. The client will be satisfied if the customer wants to buy it, but he is not the most authoritative source for the customer's desires. See Cooper [Coo99] for learning the user's desires in COTS software production.

Step 5. Writing a requirements specification

The purpose of this step is to write a draft version of the requirements specification. Some requirements may change, as a result of discussing the draft with relevant persons – but in order to engage in such discussions, you need a good document.

There are a number of different things to consider when you write the first full version of your requirements specification. This section is split into four subsections, treating separate concerns:

5.1 What should be the contents of a requirements specification,

5.2 Specification techniques,

5.3 Readability and linguistic issues,

5.4 Quality check.

Product – What do you write down?

A complete, well-structured, readable requirements specification.

Follow-up – What do you do with this document?

Send this document to relevant stakeholders. You may ask them for written comments or discuss the document with them. The latter is more work but yields better results.

5.1. Contents of a requirements specification

Way of thinking – What are the essential questions?

- Which subjects should be covered in the requirements specification?
- How to structure the requirements specification?

Approach – How to find answers to these questions?

In addition to a list of requirements, a requirements specification gives some information about the reasons for the project, the context of the system, and any other issue for which you find it relevant to provide written details. Examples of real requirements specifications are given by Lauesen's [Lau02], Chapters 11–15. A detailed, generic table of contents for a requirements specification from the Volere method [RR99] is given in Appendix D. You can use it as a checklist of things you'd like to discuss in your requirements specification. You don't want to cover all of these (unless you're doing requirements for a multi-million Euro project), so you should think about what is relevant for your project.

5.1.1 Free form or template?

Some organizations that do a lot of software projects have their own template for requirements specifications, with a fixed table of contents. Using such a standardized format has the advantage that it is easier to find particular pieces of information (if both the writer and the reader are familiar with the standard). The disadvantage is that the prescribed table of contents is probably not the most suitable for the particular project you're working on. Kovitz [Kov98] advocates the principle "form follows content." If you know what you want to say, then choose the structure that is best suited to express what you want to say.

5.2. Specification techniques

Way of thinking – What are the essential questions?

- Which parts/aspects of the environment and the desired solution need to be specified in some detail?
- What is the most appropriate specification technique in this context?

Approach – How to find answers to these questions?

Diagrams are more precise and less ambiguous than words. It is not uncommon to include use case diagrams in the functional requirements and to use a class diagram for specifying data requirements for a system. It could make sense to use an entity-relationship diagram to specify the *environment* of the system and a context-diagram to specify the interaction of the system with its environment.

What is useful depends on the project – and to a certain extent on the requirements analyst. Techniques you are familiar with work better (if they are appropriate) than techniques you have never used before. The courses Information Systems (212010) and Requirements Engineering (232081) provide enough technical background for bachelor students. Master students Business Information Technology could also apply techniques from Specification of Information Systems (233030).

5.3. Readability and linguistic issues

Way of thinking – What are the essential questions?

- Who is my target audience? Can they understand it?
- Can the presentation be improved?
- Can the text be shortened?

Approach – How to find answers to these questions?

The purpose of the document you are writing is to communicate its contents to other interested parties. In order to achieve that purpose, it pays off to make an effort to make the document well-written and well-structured. Unfortunately, the form that is easiest accessible for the target audience is not the easiest one to write. Some tips are given below.

5.3.1 Keep it short[5]

Many requirements specifications are longer than necessary. This has several disadvantages. Firstly, the readers may not read the whole document. If it's long, people are inclined to browse through the document, rather than read it. Secondly, it is more difficult to find back some piece of text. This makes it harder to use it as a reference. Thirdly, a longer text is more difficult to comprehend than a short one. Unfortunately, writing a short text is more difficult than writing a long text.

[5] Sections 5.3.1-3 are primarily based on Kovitz [Kov98] and translated from a version in Dutch compiled by Emile de Maat.

Repetition

A prime way to make a text longer than needed is to repeat information. Occasionally it is useful, to repeat text, e.g. when you give an overview or an example. Most other repetitions are not needed and can be discarded.

Metatext

Metatext is text about the text. Again, in some cases this is useful. It makes sense, for example, to explain the structure of the document in the introduction. A typical example of superfluous metatext: "In this chapter the user interface requirements are given" as introductory statement in a chapter "User interface requirements".

Generalities

Generalities are pieces of text that are not specific for the requirements that you are writing, but are more generally applicable. Consider, for example, a requirement

> *Each input screen shall fit entirely within the window and shall use as little scrolling as possible to display and/or retrieve information.*

A good user interface designers knows this and will try to apply it. A requirements specification is not a proper place to teach others about good user interface design.

Useless additions

Sometimes authors add extra text that carries no additional information. They do so, apparently, for fear of short texts – perhaps they are afraid that somebody will judge these texts as insufficient because they are short. For example:

> *The system should be user-friendly and have a simple user interface*

The second part is redundant.

Another useless addition is upgrading a short piece of text to a separate section. E.g.

> *3.3 Performance*
> *Downtime should be limited to one day per year.*

If this is all there is in Section 3.3, it could have been merged with another section.

The use of a template with a standard table of contents leads to sections like 3.3 above or, even worse,

> *3.4 Hardware constraints*
> *There are no hardware constraints*

5.3.2 Keep it simple

Requirements specifications often are hard to understand. Usually this is not because the requirements are inherently complicated, they are just specified in a complicated way. We discuss some causes for this.

Use short sentences

Many authors write too long sentences. This is often caused by the desire to provide complete and precise information. It is good to be aware, however, that all this information does not have to be captured in a single sentence. Long sentences can be made more understandable by dividing them into smaller sentences. For example

> *In this document the requirements are given for a system that Wertor will design for Myriad.*

This not a really complicated sentence. But it could be replaced by

> *Wertor will design a system for Myriad. This document gives the requirements for this system.*

Use clear and consistent terminology

When you elicit requirements, different persons may use different terms to describe the same concept. This can easily be carried over into the requirements specification, but it is confusing for the reader. It pays to make the extra effort to ensure consistent terminology. Make a glossary and make sure that the text is consistent with your glossary.

Also, the author may use a term that is known to his professional colleagues (or even worse, invent a new term) but not understood by the readers of the document. If you *must* use an unfamiliar term, make sure that you define it.

Avoid overspecification

Requirements should be complete and unambiguous. This is generally true, but it can be carried too far. Consider the following requirement for an inventory system

> *Every object in the store that is meant for sale has a unique identification code*

The store contains objects that are not for sale: shelves, fork-lift trucks, etc. These do not need a unique ID in the inventory system, but in the domain of inventory systems that is quite obvious. Hence the following, easier requirement will do

> *Every object in the store has a unique identification code*

5.3.3 Structuring text

The structure of a document can contribute a lot to its readability. Structure tells the reader what to expect where, and helps him understanding the text. In a well-structured document it is easy to find back pieces of information. This makes it suitable as a reference document.

Structuring a document is done in three steps

1. Make a list of all subjects to be treated
2. Group these into coherent groups
3. Decide upon an order in which to present them.

Most difficult is step 2. There are different ways to group subjects, and usually each of them poses some problem for presenting them in a linear order. Choose the grouping that seems most suitable and solve the ordering problems by appropriate cross-references. Make sure that you always treat one subject at the time.

Examples of different structuring principles:

- Group requirements by type of requirement
- Group requirements by stakeholder
- Group requirements by subsystem.
- Group requirements by priority, first state the “must”, then the “should”
- Order the subjects from general to specific
- Order the subjects from important to unimportant

- Order the subjects from easy to difficult, so that the reader can increase his understanding along the way.
- and so on ...

Whatever structure you choose, it is important that you support it in text and lay-out.

5.3.4 Presenting information

Whatever specification techniques you have used, there will be a lot of natural language in the document. If this contains factual information, it is advisable to present this in the form of lists and tables. Lists offer more structure, and people can use them as checklists.

A table is in fact a two-dimensional list. Information suitable for a table is hard to present in flat text. Tables are easier to read, but also easier to write.

5.4. Quality check

Way of thinking – What are the essential questions?

- Are all requirements unambiguous and complete?
- Is there a fit criterion for each requirement?
- Do we know for each requirement why it is in the specification?
- Are there conflicting requirements?
- Is the document as a whole properly finished?

Approach – How to find answers to these questions?

Below you find some quality criteria that should be applied to each requirement to determine whether it is a good requirement.

Finally, before you deliver the document, make sure that there are no loose ends, that cross-references are correct and that spelling errors, typos, and word processing errors have been eliminated.

5.4.1 Quality criteria for individual requirements

Robertson and Robertson [RR99] say that any requirement that does not satisfy all the quality criteria is, at best, a *potential* requirement. In the final version of the specification there should not be a single requirement of insufficient quality. But what we are working on here is still a draft version. For a draft version, it could make sense to include potential requirements – with an annotation of the defects yet to be solved – if these requirements were raised and should not be forgotten.

Complete?

In step 4.6 we introduced Volere's Requirements Shell [RR99], a template to be filled in for each requirement, see Appendix C. Are any components for the template not filled in? Perhaps there is nothing to fill in. For example, if there are no supporting materials, then the Shell should say "Supporting materials: None" (rather than leaving it blank). Other things might not have been clear at the time the requirement was elicited. For example, at the moment you don't know about dependencies or conflicts. Or perhaps you would need the customer to assess (dis)satisfaction values but you didn't have a chance to talk to him after the requirement was raised. It is likely that you do not yet have a fit criterion for each requirement.

If some of the questions cannot be answered right now, we have to live with that for the time being. You could indicate in the document specifically to which questions you still need answers. What you should never do is *guessing* the answers in order to complete the specification.

Precise, unambiguous and meaningful to all stakeholders?

Check whether the requirements can be misunderstood and interpreted differently from what you wanted to say.

Could possible ambiguity be reduced by stating more precisely what you mean? For example

> *"Supporting Material: Information plan of company X"*

is unambiguous only if there is a single version of this information plan. Therefore

> *"Supporting Material: Information plan of company X d.d. 22 December 2005"*

is better.

Consistent terminology (see 5.3.2) is a precondition for precise, unambiguous and meaningful requirements.

Fit criterion?

Does each requirement has a fit criterion (see 4.7), i.e. it is possible, when the system will be delivered, to establish objectively whether the requirement has been satisfied?

Relevant to the system's purpose?

Sometimes people get great ideas about what a system could also do. In the mission statement we have clearly laid down the purpose of the system. If a requirement does not contribute to the purpose, it is in the nice-to-have ("could") category. If it is included in the requirements specification, it must be made clear that it is not an essential requirement.

Unnecessary requirements are typically those with high customer satisfaction rating and low customer dissatisfaction rating.

Viable within constraints?

Does the project have the time and budget to satisfy the requirement? If not, it's not a good requirement, and should be discarded. (Or the time and budget should be adapted. If neither is acceptable the project should probably be abandoned!)

5.4.2 Consistency across requirements

In 5.4.1 and 5.4.2 we have scrutinized each requirement individually. Similar questions can be asked about the whole set of requirements. There could be

- redundant requirements;
- incompatible requirements (i.e. it is not possible to satisfy all at the same time);
- missing requirements.

Obviously there is no fail-safe way to discover missing requirements. An important way to get these is to get feedback from relevant stakeholders on the draft requirements specfication (see 6.1, validation). However, there are some consistency checks that you can do before the draft specification is finalized.

All tasks / use cases covered?

If there are task descriptions or use cases for the system, check that all actions have been covered.

System administration and support covered?

Most computer systems offer two kinds of functions: primary functions that serve the purpose of the system (users can do something useful) and secondary functions to allow the system to be operated (e.g. adding new users, maintaining the system's data). Are these secondary functions covered?

CRUD check

If there is a data model, check whether each attribute is Created and Read, and, if applicable, can be Updated and Deleted.

5.4.3 Have you finalized the document?

There are various natural roles that people can have when they work in a team (called *Belbin roles*, after the person who discovered them). Experience shows that the role *completer/finisher* is poorly represented among our students. Before submitting a document, such a person would scrutinize every detail to make sure that

- everything is numbered correctly,
- cross-references are correct,
- figures and tables appear in the right place,
- citations and references are marked appropriately in the text
- literature references in the reference section are complete,
- the lay-out is consistent,
- the names of the author(s) and other contributors are mentioned appropriately, and
- the document carries the right date and version number

If you have no such person on your team, or if you are working alone, you should force yourself to do this. This gives the document a professional appearance.

Step 6. Validating a requirements spec

The purpose of this step it to ensure that a solution that satisfies the requirement specification achieves the goals laid down in the mission statement.

Way of thinking – What are the essential questions?

- Does the specification reflect the desires and needs of the stakeholders?
- Do the stakeholders agree on the priorities, when there are conflicting requirements or when not all requirements can be met?
- Is it technically possible to meet the requirements?
- Which requirements have not passed the quality test?

Approach – How to find answers to these questions?

Validation means that you make sure that you have specified the right solution, i.e. that a product satisfying these requirements will meet the goal that was laid down in the mission statement. The persons who can decide that are the stakeholders, not the requirements analyst. (And In order to decide that, they have to be able to understand the draft specification – that is why we spent so much effort on step 5).

In situations where a complex and technically challenging system is proposed, it is wise to consult the software architects who will be involved in the design. The can warn you about requirements that are hard or impossible to realize.

If there are conflicting requirements, or if not all the requirements can be met, tough decisions have to be made. There are two things you can do: engage some stakeholders in ranking essential requirements according to importance, or ask the client to decide (or one after the other).

At the same time, when you are going back to the stakeholders with the draft requirements specification, this could be an opportunity to elicit missing elements of the requirements shell, e.g. fit criteria. You can put gentle pressure on them by explaining that, ultimately, an incomplete requirement cannot be included in the final specification.

Product –What do you write down?

The final version of the requirements specification.

Follow-up – what do you do with this document?

Deliver the specification. The requirements analysis has been completed.

6.1 Requirements validation

There are several way in which you can get feetback on the draft requirements specification. You can circulate the specification to the stakeholders and discuss it with each stakeholder individually, or you can organise a validation meeting.

If you want to know what people really think about the requirements specification, you must make sure that they understand it. That is why it is worthwhile to make the draft spec a complete, legible, accessible document, rather than circulating a premature version.

If a prototype was made for requirements gathering, you could show (an updated version of) the prototype in addition to the specification document.

Validation meeting

At a validation meeting, a selection of relevant stakeholders is present. The participants at this meeting must have enough knowledge of the application domain and the context in which the system is going to be used (the organisation for which the system is developed). Also at least one end user must be present.

The purpose of a validation meeting is to draw up a list of problems with the requirements specification, and possibly an agreed list of actions to address these problems. (It is *not* the purpose of the meeting to solve the problems here and now).

See Kotonya and Sommerville [KS89, Chapter 4] for a more elaborate description of validation meetings.

6.2 Requirements prioritization

Sometimes it's impossible to satisfy all the requirements. A finite budget is the most mundane and the most frequent reason to scale down your desires. But it could be the case that requirements are at odds with each other. Higher security may imply lower user-friendliness, and reversed. Also, if you buy an existing system or a COTS product, you have to choose from what is available, which may not be exactly what you want.

Section 4.5 discussed the MoSCoW classification and customer (dis)satisfaction values. These are absolute values, to give a first indication of what is important. When it comes to making tough decisions – what to discard, or to postpone to a future release – absolute values aren't good enough (usually too many things are important).

What is needed, then, is to assign priorities. These are relative values: is a requirement A more important or less important than requirement B?

In order to reach an optimal decision, one should

- establish relative values for all requirements
- estimate the cost of implementing the requirement

A formal method for this, based on the Analytic Hierarchy Process (AHP), is presented by Karlsson and Ryan [KR97].

Such a method yields an optimal decision, if the costs estimates are accurate and if there is no disagreement among the stakeholders (or if only the prime stakeholder, the client, matters).

When different stakeholders with different desires are important to a project, there is a political element in prioritizing requirements. When some get all their priorities granted, and others get none, the project is in for trouble.

Informal ways to assign priorities include

- Ask persons to assign a total of 100 points to different requirements in any way they want. (could be done by different stakeholder representatives as a starting point for a meeting to decide the priorities)
- Get a meeting of stakeholder representatives to agree on the 10 most important requirements. (If politics are really troublesome this could be done without further ranking among the top 10).

Step 7. Maintaining the requirements specification

The purpose of this step is to ensure that in all steps of the system's life cycle there is an accurate requirements specification for the current version of the system.

Way of thinking – what are the essential questions?

- How do you manage new requirements that arise during system development?
- How do you maintain requirements traceability and keep the requirements specification consistent when requirements change?

Approach – How to find answers to these questions?

In nearly all cases where students do a requirements analysis, the students are no longer involved in the later stages of project development. Chances are that you will not be asked to maintain the requirements specification that you delivered. Nevertheless we briefly mention some issues, completing the requirements specification life cycle.

7.1 Requirements evolution

In the ideal case, all stakeholders agree that your final requirements specification accurately descibes their requirements for the new system – at this moment. There are many reasons why requirements may change in the future:

- Testing and operation of the system may reveal defects. That is, some essential requirements were missed after all.
- Stakeholders may come up with new desires for additional features.
- The world changes, which my lead to new business requirements, or may require the system to interact with new (versions of) systems in its environment

While the system is still under development, some care should be taken in allowing new requirements to come up. *Goldplating* is a well-known software engineering risk: additional requirements continue to be added, where each requirement in itself may seem harmless, but the overall result is that it becomes impossible to build the system on time and within budget. A related risk is *feature creep*: at little extra effort (so it seems) a function can be added that would be nice to have. This may lead to a system with more capabilities than required – but at a later date and with a higher cost.

On the other hand, errors will be found and unforeseen circumstances may demand new requirements. In order to balance these concerns, any large project will have an explicit procedure for handling change requests.

7.2 Traceability

Traceability supports the maintenance of a system. The (evolving) requirements specification should on the one hand reflect the business needs and stakeholders' demands, and on the other hand specify the system's behaviour. This leads to 4 traceability relations:

- *From business/stakeholders to requirements*: It should be verified that the business goals of the system are covered. Essentially, the requirements should enable the mission statement (see 3.1) to be fulfilled.
- *From requirements to business/stakeholders*: For each requirement, there should be a business reason why the requirement is included in the specification (otherwise the requirement should be deleted).
- *From requirements to system:* For each requirement it should be known which pieces of code / parts of the system make sure that the requirement is satisfied
- *From system to requirements:* For each piece of code / part of the system it should be clear which requirements depend on it. (otherwise, it serves no purpose).

Hence, if a change is proposed, it can be easily determined which parts of the system are affected and what the effort will be to implement the change.

For any sizeable project, a specialzed tool, e.g. DOORS[6], is needed for implementing traceability.

Currently, traceability is not used a lot in practice, because it brings additional cost in the development phase, whereas most of the savings take place in the maintenance phase. (Note that on average maintenance accounts for 70 % of the total software life cycle costs). However, in future it may become a standard practice in software engineering, due to new legislation. Quality standards like the higher CMM levels enforce traceability.

[6] http://www.telelogic.com/corp/products/doors/

Glossary[7]

Client. The person who pays for the development of the system. (see also *customer*)

Constraint. A global requirement that restricts the way the system can be produced. The project budget is an example of a constraint. Usually constraint are not subject to negotiation.

Customer. The person who buys the system. This could be the same as the client. If the product is to be sold, the customer and client are different.

Data requirement. A specification of the kind of data and the relation between data elements to be stored in the system.

Business-level requirement. A description of a task to be supported by the system, without specifying what exactly the system will do.

External goal or **Client's goal.** Something the client hopes to achieve as a result of the project. The project carries no responsibility for an external goal. Nevertheless, if the external goal will not be achieved, the client may consider the project a failure. (see also *project goal*)

Fit criterion. A quantification or measurement of a requirement such that it is possible to determine whether a system satisfies this requirement.

Functional requirement. Something that the system must do, a description of the behaviour of a system

Goal. See *external goal* and *project goal*.

Migration. The path of change leading from the current situation to a new situation, in which a new system is deployed and effectively used.

Problem. A difference between what is experienced and what is desired.

Project. Throughout the text it is assumed that there is a project to deliver some system, and you are doing the requirements analysis for this project.

Project goal. Something that should be realized by the project (and for which the project manager can be held responsible). (see also *external goal*)

System-level requirement. A desired property of the system. In previous times, *requirements* was considered to be equivalent with *system-level requirements*.

Quality requirement. An overall property of the system, describing how well the system performs its functions.

Requirement. See *contraint*, *data requirement*, *functional requirement*, *quality requirement*.

Requirements process. The part of system development in which people attempt to discover what is desired.

Solution. A way to reduce a problem.

Stakeholder. Someone who gains or loses something (could be functionality,revenue, status, compliance with rules, and so on) as a result of that project.

[7] Some definitions are taken directly from other sources ([AR04], [GW89], [Lau02], [RR99]). References are given where a term is introduced in the text.

References

[BCN92] C. Batini. S. Ceri and S.B. Navathe (1992). *Database Design: An Entity-Relationship Approach.* Benjamin/Cummings.

[Ale03] I. Alexander. Stakeholders – Who is Your System For? http://easyweb.easynet.co.uk/%7Eiany/consultancy/stakeholders/stakeholders.htm

[AR04] I. Alexander, S. Robertson (2004). Understanding Project Sociology by Modeling Stakeholders. *IEEE Software*, January/February 2004.

[Cro89] N. Cross. (1989). *Engineering Design Methods.* Wiley, Chichester, UK.

[Che81] P.B. Checkland (1981). *Systems Thinking, Systems Practice.* Wiley.

[BH98] H. Beyer and K. Holtzblatt (1998). *Contextual design: Defining Customer-Centered Systems.* Morgan Kaufmann.

[Coo99] A. Cooper (1999). *The inmates are running the asylum*. Macmillan Computer Publishing, Indianapolis, IN.

[GW89] D.C. Gause, G.M. Weinberg (1989). *Exploring Requirements: Quality Before Design*. Dorset House, New York, NY.

[HC88] J.R. Hause, D. Clausing (1988). The house of quality. *Harvard Business Review,* 66(3), 63–73.

[KK92] K.E. Kendall and J.E. Kendall (1992). *Systems Analysis and Design*. Second edition.Prentice-Hall.

[KR97] J. Karlsson, K. Ryan. A Cost-Value Approach for Prioritizing Requirements. *IEEE Software* 14(5), 67–74.

[KS98] G .Kotonya and I. Sommerville (1998). *Requirements Engineering.* Wiley, Chichester, UK.

[Kov98] B.L. Kovitz (1999). *Practical Software Requirements: A Manual of Content and Style*. Manning Publications, Greenwich, CT.

[Lau98] S. Lauesen (1988).*Software Requirements: Styles and Techniques*. Samfundslitteratur, Frederiksberg, Denemarken.

[Lau02] S. Lauesen (2002). *Software Requirements: Styles and Techniques*. Addison-Wesley, Harlow, UK.

[Lun81] M. Lundeberg, G. Goldkuhl and A. Nilsson (1981). *Information Systems Development: A Systematic Approach*. Prentice-Hall, Englewood Cliffs, NJ.

[Mac96] L. Macaulay (1996). *Requirements Engineering*. Springer Verlag, New York, NY.

[RE95] N.F.M. Roozenburg and J. Eekels (1995) *Product design: Fundamentals and Methods.* Wiley, Chichester, UK.

[Ret94] M. Rettig (1994). Prototyping for tiny fingers. *Communications of the ACM*, 37(4), 21–27.

[RR99] S. Robertson, J. Robertson (1999). Mastering the Requirements Process. Addison-Wesley, Harlow, UK.

[Wie03] R.J. Wieringa (2003). *Design Methods for Reactive Systems: Youdon, Statemate, and the UML*. Morgan Kaufmann Publishers, San Francisco, CA.

Appendix A. Context-free questions

When you first enter an organization for which you are to do requirements work you may be overwhelmed by the number of potentially relevant people, departments, systems, goals and problems. This appendix lists some simple questions that you can always start with. They are called "context-free" because they apply to all kinds of problems, independent of the particular problem context. The following list is largely from Gause and Weinberg [GW89]. The problem identification and analysis questions are from ISAC [Lun81].

The business

- What kind of business is this?
- What is the structure of the business?
- Which departments of the business are involved in the system?
- What are the mission and goals of the business and its relevant departments?
- Are there any related projects?

Problems

- What are the problems?
- For each problem:
  - What is the real reason for wanting to solve this problem?
  - Can a solution to this problem be obtained elsewhere?
  - Which organizational goal is served by solving this problem?
  - How bad is the problem? (Quantify if possible)
  - How urgent is it?

Stakeholders

- Who are the stakeholders?
- For each stakeholder:
  - What is his/her relation to the system?
  - What are the responsibility relations between the stakeholders?
  - Who is responsible for improving the system?
  - Is management committed to improving the system?

Problem analysis

- Which stakeholders have which problems?
- For each stakeholder/problem combination:
  - How much is it worth to this stakeholder to solve the problem?
  - How bad is it for the stakeholder if the problem is not solved?
  - How urgently should this problem be solved?
  - How bad is it if this problem is solved one year later?
  - What is the trade-off between time and value?

The current system

- Who is using the current system and in support of which business activity?
- What problems are solved by the current system? For whom?
- What problems are introduced by the current system? For whom?
- Does the system fit into the business strategy?
- Is the system mission-critical?
  - How bad is it if the system breaks down?
- Does the system interface with legacy systems?

Appendix B. Requirements elicitation techniques

During requirements work, you must find the goals, desires and wishes of the stakeholders. This appendix lists some techniques that you can use for this.

It is important to distinguish requirements elicitation from requirements creation.

Finding out about current environment and its goals, and about the current system.

The following techniques are useful for fact-finding. They are closer to elicitation than to creation.

- **Interviews.** Asking stakeholders what they currently do and how they would like to change this. Kendall and Kendall [KK92] give a useful introduction to interview techniques for information analysis.
- **Observation** of current work. Observing what stakeholders actually do, as opposed to what they *say* they do. Beyer and Holtzblatt [BH98] give an excellent survey of models to make when observing stakeholders at work (models of flow, sequence, artifacts, culture and the physical situation), how to make them and how to create requirements from them.
- **Participation** in current work to actually experience what the current environment does. There is no literature on this: Just join the stakeholders in doing their work. Take your time doing this.
- **Questionnaires.** Sending out forms with questions to stakeholders about the current environment. Kendall and Kendall [KK92] give a useful introduction to the construction of questionnaires for information analysis.
- Study **current system documentation.** There is no literature on this. Brace yourself to digest a mountain of information.
- Study **current forms** (paper forms, screen forms). Analyzing forms in use by the current system to discover data structures and work procedures hidden in them. Batini, Ceri and Navathe [BCN92] give a useful introduction to uncovering data structures from forms.

Problem Analysis

The following techniques help you to analyze problems identified during fact-finding.

- **Soft Systems Methodology (SSM).** A method defined by Checkland [Che81] to analyze exceptionally vague problems (problems where the problem is that the problem is not known). Macaulay [Mac96] gives a handy introduction.
- **Stakeholder analysis.** Set off stakeholders against problems and analyze each problem on severity (quantify!) and urgency. Gause and Weinberg [GW89] give useful hints.

Creating requirements for new system

The following techniques can be used to create new ideas about possible solutions to problems.

- **Brainstorm.** Generating wild ideas in a group without criticizing any idea, followed by a rationalization of the ideas. Roozenburg and J. Eekels [RE95] give a very useful introduction to brainstorming for product design, including its variations, such as brainwriting (in which participants anonymously submit their ideas in writing).
- **Focus groups.** Let a group of users discuss requirements with each other. Macaulay [Mac96] gives a short introduction to the use of focus groups for requirements engineering.
- **JAD workshops.** Bring stakeholders from the customer and developer sides together and let them jointly do the design. Macaulay [Mac96] gives a short introduction to the use of JAD workshops for requirements engineering.
- **Visiting similar companies.** Visit companies with similar problems to get an idea about the desirable properties of solutions to these problems.
- **Quality Function Deployment (QFD).** Maintain traceability tables that match user requirements with system requirements. Attach weights to indicate priorities, and indicate conflicts between requirements that. Discuss with all stakeholders and agree on choices based on this traceability information. Hausaer and Clausing [HC88] give a good introduction and Macaulay [Mac96] provides a very short summary.
- **Goal-means analysis.** Make a goal tree. Indicate for each requirement the goals that it serves, and indicate for each goal the desired system properties that would help reaching that goal. Lauesen [Lau98] gives an example.

Techniques for refining system requirements and corresponding environment models

The following techniques all assume that you alrerady have some idea about system requirements and allow you to improve them.

- Collecting **supplier information**. Collect documentation from suppliers, let them give demos in order to get an idea of which system requirements can actually be realized with current commercially available technology.
- **Throw-away prototypes.** Constructing a software system that implements a few of the system requirements, and letting users experiment with it to give them the occasion to form more concrete ideas about what they really want. After experimenting, the improved

requirements are written down and the prototype is thrown away. Any software engineering book contains a section about throw-away prototyping. Ince [Inc92] is one of the many overviews. Less well-known is a description of low-tech prototyping, involving pencil, paper, glue, and role playing, described by Rettig [Ret94], that in many cases is more efficient and at least as effective as high-tech prototyping.

- **Pilot project.** Implement the system in a part of the organization where it is not critical, in order to get experience with real use of the system. This should lead to improved requirements.

Appendix C. Volere Requirements Shell

In the Volere method [RR99], Suzanne and James Robertson give a template to be filled in for each requirement – see Figure C1. They call it the Requirements Shell. It is suggested that you carry copies of the template with you when go around gathering requirements.

Filling in the template for each requirement reminds you of what you want to ask the person(s) you're talking with. The slots have the following purpose

- *Requirement #*: unique ID for each requirement
- *Requirement type:* constraint / data / functional / quality
  (or refer to section in requirements specification template in Appendix C)
- *Event/use case # :* If use cases or an event list has been specified, refer to its number
- *Description:* A one-sentence statement of the intention of this requirement
- *Rationale:* Why is this requirement considered important or necessary?
- *Source:* Who raised this requirement?
- *Fit criterion:* A quantification of the requirement used to determine whether the solution meets the requirement (not always easy to determine up front. If no sensible criterion can be found when the requirement is raised, we suggest to leave it open for the time being.)
- *Customer (dis)satisfaction:* Measures for the (un)happiness of the customer if this requirement is (not) implemented. See section 4.5
- *Dependencies:* Dependencies between this requirement and others.
- *Conflicts:* Requirements that contradict this one
- *Supporting Materials:* Pointer to supporting information
- *History:* Changes to this requirement (and reasons why)

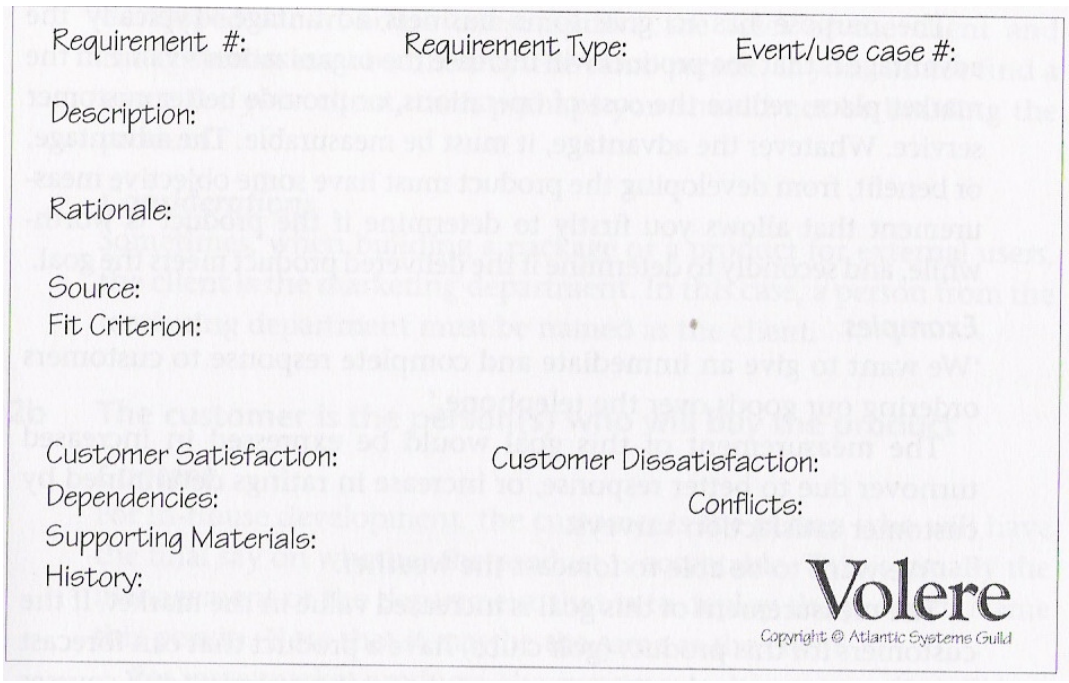

Figure C1: Volere Requirements Shell

Appendix D. Volere Requirements Specification Template

The Volere method [RR99] provides a template for the contents of a requirements specification. Here we only give the contents with some bits of explanation. An extensive description of the template can be downloaded from www.volere.co.uk. It is very thorough and complete, and for a small project there is probably no need write a requirements specification with 27 chapters. But you may use this as a checklist.

Project Drivers

1. **The purpose of the project**
2. **Client, customer and other stakeholders**. The client is the person paying for the development, and owner of the delivered product. The customer is the person buying the software. Client and customer are the same for in-house developments but different when the system to be developed will be sold to others.
3. **Users of the product**

Project Constraints

4. **Mandated constraints**. Constraints that the project must satisfy. Includes development time and budget.
5. **Naming conventions and definitions**
6. **Relevant facts and assumptions**

Functional requirements

7. **The scope of the work**. Describes the domain. Could include a context diagram.
8. **The scope of the product**. Could include use case diagram.
9. **Functional and data requirements**

Non-functional requirements

10. **Look and feel requirements**
11. **Usability and humanity requirements**
12. **Performance requirements**
13. **Operational requirements**. Expected physical environment, hardware, and software applications with which the system should interface.
14. **Maintainability and support requirements**
15. **Security requirements**
16. **Cultural and political requirements**
17. **Legal issues**

Project issues

18. **Open issues**. Issues that have been raised and do not yet have a conclusion.
19. **Off-the-shelf solutions**. Ready-made software products or components that can be used
20. **New problems**. Problems that may result from introducing the system.
21. **Tasks**. A stepwise description of system development, delivery, and implementation
22. **Cutover**. Issues related to the migration to the new system.
23. **Risks**
24. **Costs**
25. **User documentation and training**
26. **Waiting room**. Requirements that will not be part of the agreed system, but could be included in future versions.
27. Ideas for solutions