Problem-Oriented Requirements in Practice – a Case Study

Soren Lauesen

IT University of Copenhagen, Denmark
slauesen@itu.dk

Abstract. [Context and motivation] Traditional requirements describe what the system shall do. This gives suppliers little freedom to use what they have already. In contrast, problem-oriented requirements describe the customer's *demands*: what he wants to use the system for and which problems he wants to remove. The supplier specifies how his system will deal with these issues. The author developed the problem-oriented approach in 2007 on request from the Danish Government, and named it SL-07. **[Question/problem]** SL-07 has been used in many projects – usually with success. However, we had no detailed reports of the effects. **[Principal ideas/results]** This paper is a case study of SL-07 in acquisition of a complex case-management system. The author wrote the requirements and managed the supplier selection. Next, he was asked to run the entire acquisition project, although he was a novice project manager. Some of the results were: The problem-oriented requirements were a factor 5 shorter than traditional requirements in the same domain. Stakeholders understood them and identified missing demands. Suppliers could write excellent proposals with a modest effort. The requirements were a good basis for writing test cases and resolving conflicts during development. The delivery was 9 months late, but this was not related to the requirements. **[Contribution]** This is a publication of a full, real-life, complex requirements specification, the selection document, error lists, etc. The full texts are available on the author's web-site. The paper discusses the results and illustrates them with samples from the full texts.

Keywords: Problem-oriented requirements, SL-07, COTS-based, case study, supplier selection, issue resolution, fixed-price contract, usability requirements.

1 Background

Requirements can be written in many ways: traditional system-shall requirements, various kinds of use cases, user stories, UML-diagrams, etc. Does it matter which kind of requirements we use, e.g. which of the many kinds of use cases or user stories we use? It does. It influences whether stakeholders can check that requirements cover their needs (validate them), suppliers can provide meaningful proposals, the parties can agree whether issues are bugs or requests for change, etc. The author has seen many real-life requirements specifications and published five very different ones in his textbook (Lauesen [5]), where he also explains the consequences of each kind of

requirements. Some of the consequences have been disastrous, such as losing a business opportunity of 100 M$ because of traditional system-shall-requirements.

What does research say about the way we write requirements? Amazingly very little. Publications rarely provide examples of real-life requirements, and how they worked in practice. Many papers have statistics and general discussions of requirements (e.g. Nurmuliani et al. [13] about requirements volatility), but the reader wonders what the real requirements looked like. As another example Bruijn and Dekkers [1] investigated how many requirements in a specific project were ambiguous and how many of them actually caused problems. However, we don't see any of the requirements, not even the one that caused serious problems. Maiden and Ncube [11] wrote about acquisition of a COTS system and gave advice on how to do it better. Here too, we don't see any requirements. Even in textbooks about requirements, we rarely see real requirements. The focus is on the requirements processes. Exceptions are Robertson and Robertson [15], who illustrate all kinds of requirements with tiny pieces, primarily from a road de-icing system, Kotonya and Sommerville [4], who show tiny pieces from a university library system, and Cockburn [2], who shows examples of many kinds of use cases. None of them show a full, real requirements specification or substantial parts of one, nor the supplier's proposal or reports of how the requirements worked in practice.

In 2007, the author published Requirements SL-07, an exemplary requirements specification for an electronic health record system with a guide booklet. It covered all kinds of requirements in a problem-oriented way: we don't specify what the system shall do, but what the user will use it for. The Danish government had requested it as part of their standard contract for software acquisitions, K02. Analysts can download it, replace irrelevant requirements with their own and reuse large parts.

SL-07 was intended for software acquisitions where large parts existed already (COTS). However, SL-07 proved equally useful for other kinds of projects, such as product development or agile in-house development.

In this paper, we show how SL-07 was used in a real-life project: acquisition of a COTS-based system for complex case management. We show how the spec developed, how the suppliers reacted, how we selected the winner, how issues were resolved during development, and why the project was 9 months late. You can download the full specification with the supplier's proposal, the selection document, the list of errors/issues, the test script, etc. from the author's web-site:

http://www.itu.dk/people/slauesen/Y-foundation.html

Method

This is a report of a real project. The project was not action research, nor planned to be part of any research. As a consultant, the author had helped many customers with requirements, but left project management and acquisition to the customer. The Y-Foundation project started in the same way, but developed into the author being also the project manager. Later he got permission to anonymize and publish papers from the project. This paper is based on 795 emails, other existing documents, discussions and meetings that the author participated in. In addition, the author later contacted the new foundation secretary and the supplier to get their view on the system after more than two years of use. The documents have been translated from Danish and anonymized.

There is an obvious validity threat since the author reports about a project where he had a significant influence. The threat is reduced by giving the reader access to the original documents, which were shared with stakeholders and suppliers. However, it has not been possible to anonymize the emails.

2 The Y-Foundation Case

Twice a year the Danish Y-Foundation (synonym) receives around 300 applications and gives grants to some of them. There are two grant areas: Engineering and Medical. The Foundation has two full-time employees (a secretary and the CEO) and two part-time (an accountant and a web-editor). The board of the Foundation has four members - two business members and two domain experts, one in engineering and one in medicine. All board members look at all the applications. At a board meeting, the board decides which applications to grant. Next, it is a clerical task to send accepts or rejects to the applicants, pay grants and receive final reports.

The entire process was manual. The applications were paper documents. They circulated between the board members prior to the board meeting. The secretary maintained a spreadsheet that gave an overview of the applications.

In January 2013, the foundation decided to acquire a grant management system and a new CMS on a fixed-price contract. Applicants would upload grant applications on the foundation's web site. The board members would in parallel look at the applications and see the other board member's ratings. At the board meeting, they might modify their rating, and the other board members would see it live. After the meeting, the secretary would send bulk emails to applicants; handle payment of grants; remind applicants to send a final report, etc.

The foundation contracted with the author to write the requirements, later to handle also supplier selection, and finally to be the project manager (PM) of the entire project. He wrote and maintained the requirements based on the problem-oriented requirements in the SL-07 template [7]. Most of the system existed already. The new parts were developed in an agile way. The system was deployed March 2014 with several open issues and completed October 2014, nine months late.

3 Problem-Oriented Requirements and SL-07

Jackson [3] distinguished between the problem space (outside the computer system) and the solution space (inside the system). He pointed out that requirements should describe the problem domain, leaving the solution domain to the developers. However, it wasn't clear where the boundary - the user interface - belonged.

When we use the term *problem-oriented requirements*, we don't specify the user interface. It is part of the solution space. The developer/supplier has to provide it. We describe not only functional requirements in a problem-oriented way, but also usability, security, documentation, phasing out, etc.

Here is an example of problem-oriented requirements from the Y-Foundation. It is the requirements for how to support the board members during the board meeting.

From the board member's point of view, discussion of applications during the meeting is one task, carried out without essential interruptions. At first sight, a task description looks like a typical use case, but it is profoundly different:

Task C21. During the board meeting
This task describes what a board member does with the grant applications during the meeting.
Start: When discussion of the applications starts.
End: When all applications have been discussed for now.
Frequency: Twice a year.
Users: Board members. The four board members and the secretary look at the applications at the same time and note their own comments directly in the system. See also access rights in H1.

Subtasks:	*Proposed* solution:	Code:
1. Look at each application. See what the other board members mean, preferably live as soon as they have indicated something. Look at the full application and attached documents.	*As task C20*. [C20 shows the proposed screen with a list of applications, each with a traffic light for each board member] *The system updates the list of applications without the board members having to click a "refresh".*	
2. Record your conclusion and your private comments.	*As task C20*.	
3. Maybe record the joint conclusion.	*As task C20.*	

The text before the table is not requirements, but assumptions the supplier can make and the context in which the task is carried out. The requirements are in the table. In this case there are three requirements, each of them being a subtask of the full task. Column one shows the user's demand, what he wants to do. Column 2 may initially show the customer's idea of a possible solution, later the supplier's proposed solution. In the real document, the proposed solution is in red, here shown also in italics. Column three (the code column) is for assessment, reference to test cases, etc.

The subtasks can be repeated and carried out in almost any sequence. The user decides. A subtask could also be a problem the user has today. We might have written this "problem subtask":

1p. Problem. Today you cannot see what the other board members mean. You have to wait and hear.	*The system updates the list of applications without the board members having to click a "refresh".*	

Notice that the task doesn't describe an interaction between user and system. It describes what the user wants to achieve. The requirement is that the system supports it. We have shown experimentally that tasks perform much better than use cases in many ways, for instance in their ability to deal with the business-critical needs of the customer (Lauesen and Kuhail [6]).

User stories have become widely used. We might translate each task step to a user story. Using Lucassen [9] as a guide, step 2 would become this user story:

> *As a board member, I want to see the application's traffic lights, so that I can record my conclusion and my private comments.*

The traffic lights have now become requirements. In the task version, the traffic lights are potential solutions. This makes user stories less suited for COTS-based systems where most of the system exists already. An existing system might not use traffic lights at all, yet provide a good solution. If we replace all the task steps with user stories, we have defined a solution: a rather detailed description of the functions on the user interface. However, we cannot go the other way from user stories to task descriptions, because we have lost information about the larger context in which these user stories take place. It will for instance be hard to see which user stories should be

Table 1. The Y-Foundation requirements

Contents

supported by a single user screen. An Epic might help here, but there are no traditions or guidelines that ensure that it will group user stories in a useful way. With SL-07, grouping and context description are compulsory.

Stakeholders like user stories [10], probably because they have a simpler and more rigid structure than use cases, and have more user focus than system-shall requirements. However, there are no experience reports about how successful user stories are in fixed-price projects, how stable they are, and how many customer-supplier conflicts they resolve.

Table 1 shows the table of contents for the final SL-07 spec, including the supplier's proposal. Around 30% of the pages are tasks (Chapter C). Another 20% are descriptions of data the system must store (Chapter D). It includes a slim E/R data model and a detailed data description. Business aspects, system integration and non-functional requirements take up the rest. All requirements are written in a problem-oriented way. Around 90% of Chapters G to L can usually be reused word-by-word.

The spec contains a total of 275 requirements. Of these, 100 are task steps, 80 are descriptions of the fields in the data model (each field is a requirement). The remaining 95 requirements are system integration and non-functional requirements.

User stories and use cases cover only what corresponds to the 100 task steps.

SL-07 is not just a problem-oriented way to express requirements. It provides a convenient format that makes it easy to match requirements with the supplier's proposal, track requirements to test cases, and track business goals to requirements. It also serves as a checklist for what to remember, with realistic examples of everything. Based on experience with many projects, it has grown over the years to deal with new topics, e.g. supplier selection criteria and recently (version 5) EU's General Data Protection Regulation (GDPR).

Usability requirements

Usability is important in most projects, but it is hard to specify in a verifiable way. In the Y-Foundation, usability requirements played a major role in determining whether an issue was an error or a request for change.

The SL-07 template, Chapter I, covers usability requirements. It requires what usability specialists agree on: Make early, unguided usability tests of the user interface (or a mockup) with potential real users; redesign and test with new users until the test results are acceptable (Nielsen [12], Redish et al. [14]).

This cannot be used directly in our case where the complex part of the user interface has only one user (the secretary) and the medium complex part has only four users (the board members). We came up with these problem-oriented requirements:

Usability I1. Ease-of-learning and task efficiency

Requirements:	*Proposed* solution	Code
1. The secretary must be able to carry out the tasks in Work Area 1 without serious usability problems ["Serious" defined below the table]	With a functional version of the system, a secretary carries out examples of tasks without guidance. On the way, the secretary may ask the supplier's expert. The secretary assesses whether the system is sufficiently efficient and easy to use. *Offered.*	
2. Board members … (similar)	(similar) *Offered.*	
3. Potential applicants must be able to carry out the tasks in	A think-aloud test with three potential applicants is made. The user cannot ask when in doubt. *This*	

Work Area 4 without serious usability problems.	*is the customer's own responsibility.*	

A **serious** usability problem is a situation where the user:
a. is unable to complete the task on his own,
b. or believes it is completed when it is not,
c. or complains that *it is really cumbersome*,
d. or the test facilitator observes that the user doesn't use the system efficiently.

The first requirement (I1-1) worked well in practice. It says that the users may not encounter serious usability problems during their tasks, and it defines what a serious usability problem is.

The requirements were used in this way: During acceptance testing, the secretary carried out various test tasks. When she was stuck, we recorded it as an issue ("defect"), according to requirement I1-1. Later, the secretary sat next to a supplier specialist, carried out the tasks and asked when needed. Some of the issues were true defects; others were things we learned how to do.

The user interface for the board members was tailor-made, based on the secretary's vision and agile (iterative) development with the supplier. The user interface became intuitive to the board members, but there were many errors in the detail (bugs). They were gradually removed.

For the potential applicant's user interface, we accepted the responsibility (I1-3) and paid the supplier for changes, as we in an agile way developed the web part.

4 Elicitation and Specification of the Requirements

The PM (the author) used 11 weeks to elicit and write the requirements that we sent to the potential suppliers. He spent 40 work hours on it. A month-by-month timeline of the project with hours spent and number of emails handled, is available at the author's web site [8]. Here is a summary:

18-01-2013: The consultant (the author) started his work.

02-04-2013: Requirements version 2.4 was ready (34 pages + 3 page data examples). The requirements had been through versions 1.0, 1.1, 2.0, 2.1, 2.2 and 2.3. Each version was the result of interviews, study of existing documents, comments from stakeholders, and a focus group with potential applicants. The contents grew almost chapter by chapter according to the TOC in table 1. Chapters C (tasks to support) and D (data to record) required most of the work. The last parts from Chapter H (security) to Chapter L (maintenance) were around 90% reuse of the template example.

10-04-2013: We sent this version to the three suppliers we had selected and asked for a meeting with each of them. They should show how their system supported the requirements. They could also suggest changes to the requirements. They did not have to write anything.

06-06-2013: Requirements version 2.5 was ready (still 34 + 3 pages). After the meetings with the suppliers, we had 6 comments that we included in version 2.5. An important one was to allow other accounting systems than the present one. No ma-

jor changes were needed. We sent this version to the three suppliers asking for a written proposal. The supplier should write his proposed solution in column 2 of the requirement tables or as *solution notes* above or below the table. He should also quote the price.

28-06-2013: Contract version 1.0 was ready (44 pages). We got proposals from all three suppliers and selected one of them. His version of the requirements with his proposed solution became version 1.0 of the contractual requirements.

13-09-2013: Contract version 2.1 was ready (44 pages). During the contract work, we made a few minor changes in the contractual requirements. This is the version available at the author's web site [8]. It includes a detailed change log.

During development, we did not make further changes to the requirements. We managed errors and changes through a list of issues, as explained in the development section below. In two cases, we made an amendment to the contract.

During elicitation, we received many stakeholder comments, but we usually had to restructure them to fit them into the template. Many analysts simply make each comment a new requirement. In fact, some analysts consider requirements a list of the user's wishes. However, this leads to unstructured requirements that are hard to implement and keep track of. In addition, user wishes may be solutions that conflict with the supplier's way of doing things. In our case, we took care to translate the comments into the SL-07 style and insert them in the proper template part. Here are two of the wishes we got, the resulting requirements, and the selected supplier's proposed solution (red in the real document, italics here). In several cases, we had to add more than one requirement to meet the wish:

Wish from a domain expert

I want a "private space" for my own comments on the grant application. We translated it into a task step (functional requirement) and a data requirement. It looked like this, including the supplier's proposed solution in italics:

Task C20. Assess applications before the board meeting

Subtasks and variants:	*Proposed* solution	Code
1. Look at the applications you have to assess …	*The system shows a list of …*	
…	...	
6 [new]. Note your private comments that are not intended for others.	*Noted directly in the list.*	

Data D5. Application role [Name of a data class]

Fields and relationships:	*Proposed* solution	Code
1. roleType: …	*The customer can maintain a list …*	
…	*...*	
9 [new]. private_comment: The board member's private comments. Not visible to others.	*Yes*	

Wish from the auditor

It shall not be possible to pay money to an applicant's bank account until the account number has been approved by someone else than the one who created the account

number in the system. We needed an elaboration. He explained that he had seen fraud where a secretary handled a large grant by changing the applicant's bank account number to his own, paying the amount to it and informing the applicant that the application had been rejected. It became these two new requirements:

Task C14. Pay grants

Subtasks and variants:	*Proposed* solution	Code
1. Make a list of payments …	*The system creates the list …*	
…	*…*	
4 [new]. Check that account numbers are what the applicant specified.	*If the account number has been changed, this is clearly flagged.*	

Security H5. Protection against threats

Threats to protect against:	*Proposed* solution	Code
…	…	
5 [new]. The system must prevent that someone forges the bank account number prior to the payment.	*The system can in the payment list show what originates directly from the applicant …*	

5 Supplier Selection

In general, suppliers spend a lot of time and money on proposal writing and customer meetings, often more than 500 hours for a proposal. Making it easy for them is important for getting good proposals. In our case, the three suppliers found it easy to reply. According to their comments, a supplier spent only 20-30 work hours. There are several reasons for this.

First, the requirements were short, just 34 pages. According to the suppliers, traditional requirements in this domain are hundreds of pages.

Second, the suppliers did not have to write anything before the first meeting. They just had to present their solution and explain how it met our requirements.

Third, when they sent their written proposal, they could easily write how their system met each of the requirements, because the demand (e.g. the task step) was clearly visible. However, only two of the three suppliers did this.

Fourth, when we had received and discussed the proposal with the supplier, we took the burden of editing the proposal and sending it to the supplier before he quoted a price.

Supplier A offered a solution based on Microsoft's CRM-system (for managing communication with customers), Microsoft's SharePoint, etc. SharePoint was used also to develop the Foundation's web-site. Everything was standard components that were configured and combined. No programming was necessary.

Supplier A didn't reply to each of the requirements. He described the solution as a list of modules to be delivered, e.g. "customer management, segmentation, internal case management". We couldn't see how all of this related to the Foundation's work.

However, we had the promised meeting where we discussed their proposal. During the meeting, we managed to walk through all the SL-07 requirements, listen to the

way they planned to support them, and take notes. Next, the PM edited the notes into the SL-07 requirements and returned them as the agreed solution.

Supplier B offered a solution based on their own extensions to SharePoint, Outlook (e-mail) and either Navision or eConomic (accounting). SharePoint was also used to develop the Foundation's web site. Possibly, a bit of programming would be needed for the Foundation.

B had carefully written their solution proposal for each of the Foundation's requirements, but in several essential places they just wrote "needs more analysis". For instance, it was obscure how the accounting system would be integrated. Some solution proposals showed a misunderstanding of the needs.

Supplier C offered a solution based on their existing case management system (an extension of Microsoft's SharePoint), Outlook (e-mail), Navision (accounting) and Wordpress (Open source system for development of the Foundation's web-site). Possibly, a bit of programming would be needed for the Foundation.

C had carefully written their solution proposal for each requirement. As an example, the most important central overview screen (the list of grant applications) was shown in graphical detail. The SL-07 requirements including solutions were 44 pages. The most uncertain parts would be tested early in the project and both parties could terminate the contract if the test failed (proof-of-concept, requirements B3).

Choice: We chose supplier C based on three factors: Financial benefit, risk, and cost of product including 4-years of operation. See details on the author's web-site.

6 Development

The plan was that the system should be acceptance-tested early December 2013 and the 4-week operational test completed before New Year. Actually, full delivery didn't take place until end of September 2014 (a delay of 9 months). Here is a summary of the development steps:

18-09-2013: We sign the contract with supplier C and start development. The supplier had identified integration with the accounting system, tax reporting and automatic bank transfer as the most risky parts. He had not tried this before. The plan was to make a POC (Proof of Concept, B3) to reduce the risk. However, it turned out that the bank needed many weeks to give electronic access.

11-10-2013: We accept the POC although we have not completed an electronic bank transfer. However, the system can do all the preparatory work. The system is able to make the basic communication with the accounting system, which is supposed to handle also the tax reporting. Implementation of the applicant's parts, the board's parts, and the secretary's parts continues.

11-11-2013: According to the contract, the supplier should have completed the system test by now, but he needs just a few more days. Everything looks promising.

14-11-2013: To speed up things, we run our first acceptance test. We don't get very far. We encounter and report 23 issues (defects, mistakes, etc.).

20-11-2013: The supplier reports *system test passed*. We try acceptance testing again, but don't get much further. The list of issues grows, some issues are resolved, many remain open or are reopened. The ambition was to deploy the entire system before Christmas, where applicants become busy sending grant applications. We decide to focus on the on-line application part and delay other parts.

23-12-2013: We deploy the on-line application part. It works fine, although some applicants need assistance to circumvent system issues. At the application deadline 15-01-2014, we have 225 applications. There are now 69 issues on the list, including the closed ones.

31-01-2014: We have now been in operational test for the four weeks specified in the contract. There are only 12 open issues on the list. They seem tiny and we agree that they can be handled during the warranty period. We accept delivery and pay the supplier the full amount (around 100,000 $) plus 40 hours for changes.

25-02-2014: The system parts for the board and the secretary work miserably. Often the users have to login for each document they want to see. This is extremely cumbersome because a grant application contains several documents. Errors come and go. We focus on repairing the issues. The PM strives to postpone discussions about issues being defects or changes, to meetings in the steering committee.

27-03-2014: The great grant meeting in the board. The supplier has an expert in the room to offer support. Fortunately, the meeting is a success. Although a bit slow, everybody can see each other's vote. Earlier the board spent the whole day discussing the applications. Now they have already agreed on most of them (those with four red lights or four green lights in the list of applications). In around an hour, they deal with the applications that need discussion. They spend the rest of the meeting discussing strategic issues, which they did not have time for earlier.

15-04-2014: The secretary cannot handle the grants. There are things she doesn't know how to do and outright errors in the system. The supplier is silent. There is no financial incentive anymore. We escalate the problems to the CEO level and things move on slowly.

01-09-2014: There are still 9 open issues on the list.

01-10-2014: The last issues have been resolved or renounced. The business goals are met and the users are happy with the system.

Test cases and user manual

For the acceptance test, we developed a test script that would cover test of most of the requirements. It had one or more sections for each of the requirements sections. Here is part of the script for tasks C12 and C21:

Test script: Section 6. C12 and C21. During the board meeting

NN [Secretary] and a [simulated] board member work concurrently with the system.

1. Board member writes own public and private comments for application L and M. Votes yellow for both.
2. Check that NN and other board members can see the vote and the public comments.
3. NN records for application L: green, M: red, C: green.
4. NN records that C is worth publishing.
5. Ask the board to confirm that everything is correct. Start time monitoring, 12 hours.

In a copy of the requirements, we made the code columns refer to the line or lines in the test script that would test this requirement. Now it was easy to spot the requirements that were not tested.

User manual. As explained above, the secretary's part of the system was not intuitive. New secretaries would come aboard and would need help. Since the task part of the requirements corresponded to observable periods of working with the system, it was obvious to make a guide section for each task.

So we did. Basically, each guide section consisted of a screenshot of the situation, and for each button a callout with a short explanation of the subtask which would use it. We tested the first part of the user manual with a potential secretary. The result was that it would not suffice as a stand-alone manual, but with a bit of initial personal explanation, it allowed a new secretary to experiment on his own.

7 Error Handling and Issue Resolution

During the project, the list of issues grew to 130 (including 23 from the first test). At the end, they were all closed, i.e. resolved or renounced.

We can classify issues in this way:

1. Defect: The system violates the requirements. The supplier must cover the cost of repair. Includes serious usability problems where the system could do what the user wanted, but the user couldn't figure out how (requirements I1-1 and I1-2).
2. Failed expectation: Although not specified as a requirement, the developer should have known and must cover the cost. Includes obvious errors. Danish contract law uses this principle.
3. Change: A new or changed requirement. The developer couldn't know. The customer must pay for the repair.
4. Ignore: A mistake, a duplicate, cannot be reproduced, or the customer decides to accept it as it is.

Using these definitions, we get the number of issues shown in Table 2.

Table 2. Issues according to type

45 defects related to these requirements:	
	18 related to usability (Chapter I)
	10 to security (H)
	7 to deployment (J)
	5 to tasks (C)
	2 to data (D)
	1 to system integration (F)
	1 to response time (L1)
49 failed expectations	
22 changes	
14 ignore	
130 Total	

There are 45 defects (violated requirements). From the customer's point of view, it is an advantage that issues are classified as defects, rather than changes (for which he has to pay). More than 60% of the defects were violation of usability requirements and security requirements (H4-2 and H4-3, protecting against human mistakes). See examples below.

The 49 failed expectations can be obvious bugs or issues the supplier should know. See examples below. There are 22 change issues. The customer had to pay. Here, better elicitation might have helped (see the *discussion* section below).

Examples of defects (violated requirements)

#F8 When the user scrolls far down the application list, the list headings disappear. [Violates I1-2. It was a serious usability problem for the board. See discussion of #F8 below.]

#F13 The test person applied for 81.000 DKK, but it ended up as 81 DKK. [Denmark uses decimal comma. Violates H4-2: All data entered must be checked for format, consistency and validity.]

#25 When sending bulk emails to all rejected applicants, we need to make a few individual changes. Not possible, said the supplier. [Violates C13-11, where the supplier had proposed this solution: *The secretary can change them individually before sending them*. So he had to find a solution – and he did.]

#28 Wanted to pay an applicant. By mistake, the secretary clicked one with red lights, meaning *reject*. The system couldn't undo it. [Violates H4-3: The user must be able to correct mistakes easily.]

Examples of failed expectations

#F23 File names in the application form: Only the top half of the letters are visible. [This is an obvious error. You would not write a requirement about such details.]

#71 Port 80 must be used in the upload part of the web-site. For security reasons, many companies block other ports. [The supplier used another port, and as a result many professional applicants couldn't upload their application. We argued that port 80 was the usual default, and that the supplier had announced the solution as accessible from everywhere.]

Examples of changes

#44 It must not be allowed to upload travel applications without an Excel budget.

#70 The grant receiver's bank account should show the payment with the receiver's project ID. We showed the foundation ID only. Important for universities that receive grants for many projects. They couldn't trace the payment to a department. [We had missed this rule because we forgot to treat the receiver's accountants as stakeholders. Fortunately, the error was easy to repair.]

Example of issue resolution

#F8 Scrolling a list with headings. The board member's list of applications has a line for each of the 200-300 applications. It has 16 columns, including 5 "traffic lights", one for each member and one for the secretary. When the user scrolled down the list, the headers moved away too and the user couldn't see what was what. It was a serious usability problem. We had this dialog with the supplier:

- Supplier: It is web-based, so it is impossible to do it better.
- The PM found a solution on the web and gave the supplier the link. It *is* possible.
- Supplier: It will be costly.
- We: It is a usability defect (I1-2), so do it, please.

8 Discussion

Why was delivery late?
The selected supplier suggested developing and deploying the system in 3 months, based on his COTS system. Actually, it took 11 months. The reasons were:

1. The supplier had been too optimistic with system integration. The POC (Proof of Concept) had not revealed the complexities, partly because we had not anticipated that the bank needed many weeks to provide electronic access.
2. For a complex web application that handled also Office documents, browsers turned out to behave differently and it was hard to figure out what to do.
3. For programming, the supplier used a subcontractor without domain knowledge. This caused many misunderstandings, also because the communication path became long.
4. We had accepted the delivery and paid the supplier, assuming that the few open issues could be handled as maintenance. This removed the financial incentive for the supplier, and things went very slowly.

It is hard to see that additional requirements would have reduced the delay. Better project management would.

Would traditional requirements help?
Traditional requirements in this domain are hundreds of pages, told the suppliers. The author has experienced it himself. The university where he works wanted a case-management system for the entire university. It was a bit more complex than the Y-foundation system, e.g. because it had to handle many types of cases. The project manager and the author wrote SL-07 requirements similar to the ones for the Y-foundation. We spent around 60 hours to do this. The spec was 45 pages. However, in order to speed up the process and avoid a full EU acquisition, the university wanted to build on an existing requirements framework for case management systems, where several suppliers had been prequalified. So the university hired two consultants. They spent around 100 hours to move various SL-07 requirements into the system-shall framework. They refused to include usability requirements because *it was impossible to define usability, so forget about it.* (As explained above, usability requirements saved many troubles in the Y-Foundation case.)

The result was a requirements specification of 240 pages with lots of mandatory requirements. The contract part was an additional 120 pages. Just having suppliers send a proposal was a problem. We got two proposals, only one of which met the mandatory requirements. The conclusion is that this kind of requirements would not have helped. SL-07 doesn't use mandatory requirements because requirements can rarely be assessed in isolation. They interact. Instead you may insist on adequate support of requirements *areas*, e.g. board meetings or usability. In the Y-Foundation, the mandatory "requirement" was that the business value of the entire acquisition shall be positive (requirements section B4).

Avoiding the issues
Issue handling takes time, also for the issues where the supplier has to cover the cost. To what extent could we have prevented the issues? Let us look at the issue classes one by one:

Defects: Defects are violated requirements, so defects are a sign that requirements work well. Otherwise, the issues would have been changes at the customer's expense. But it still makes sense to prevent them. More than 60% of the defects were usability issues and handling human mistakes. You would expect that they might be prevented by early prototyping, but most of them are of a very technical nature and would not have been caught in this way. As an example, it seems unlikely that any of the four defects above would have been caught by prototyping. Would agile development help? No, all the new parts of the system were developed in an agile way.

Failed expectations: Again, most of them are very technical, and better requirements would not help.

Changes: Thirteen out of the 22 change requests were about data not being shown when needed, doubts about mandatory data fields, or confusing labeling of data fields on the user interface. If the supplier had accepted responsibility for usability of the applicant's web interface, these issues would have been defects. A more profound change was that a new application state was needed in addition to the nine specified. Fortunately, none of the changes were costly to implement (40 hours total).

Better requirements elicitation would have helped, e.g. the customer exploring the data presentation with prototypes or wireframes. The wire frames would fit into SL-07 as *solution notes*, in that way not being requirements.

COTS or tailor-made: As expected, the problem-oriented requirements were equally suited for the COTS parts and the tailor-made parts. Traditional *system-shall* requirements or user stories are less suited for COTS, because the COTS system may support the need, but not in the system-shall/user story way.

9 Conclusion

The case study has shown the following benefits of problem-oriented requirements in this project. Since there is no similar study of other ways to specify requirements, we have little to compare with. The hypothesis is that the benefits below can be expected in other projects too, if they are based on SL-07 and have an analyst with solid SL-07 experience.

1. The problem-oriented requirements were 5 times shorter than traditional requirements in the same area.
2. The requirements were well suited for COTS-based solutions, since they didn't specify what the system should do, but only what it was to be used for.
3. Elicitation and requirements writing took just 40 hours. This was due to reuse of the SL-07 template example and the way it expresses requirements, but also to the author having extensive experience with SL-07.
4. Stakeholders could understand the requirements and explain what they missed.
5. Suppliers could write excellent proposals with a modest effort (20-30 hours).
6. It was easy to select the winner because we could se what each proposal supported well and poorly.

7. The requirements were a good basis for resolving conflicts about who pays when issues came up during development.
8. They were also a good basis for writing test cases and user manual.
9. The SL-07 usability requirements and the security requirements about guarding against human errors, eliminated a lot of change requests.

References

1. Bruijn, F. and Dekkers, H.L.: Ambiguity in natural language software requirements: A case study. In: Wieringa and Persson: REFSQ 2010, pp. 233-247.
2. Cockburn, A. (2001): Writing effective use cases. Addison Wesley.
3. Jackson, Michael (2001). Problem Frames: Analysing and Structuring Software Development Problems. New York: Addison-Wesley.
4. Kotonya, G. and Sommerville, I. (1998): Requirements Engineering, Processes and techniques. John Wiley & Sons.
5. Lauesen, S. (2002): Software requirements – styles and techniques. Addison-Wesley.
6. Lauesen, S. & Kuhail, M. (2011): Task descriptions versus use cases. Requirements Eng (2012) 17:3–18, DOI 10.1007/s00766-011-0140-1.
7. Lauesen, S. (2016): Guide to Requirements SL-07 - Template with Examples, ISBN: 9781523320240. Also available at: http://www.itu.dk/people/slauesen/index.html
8. Lauesen, S. (2017): Requirements for the Y-Foundation. Full requirements specification including the supplier's reply, the selection document, and the list of errors/issues. http://www.itu.dk/people/slauesen/Y-foundation.html
9. Lucassen, G. et al. (2016): Improving agile requirements. Requirements Eng (2016) 21:383-403.
10. Lucassen, G. et al. (2016): The use and effectiveness of user stories in practice. In: Daneva & Pastor: REFSQ 2016, LNCS 9619, pp. 205-222.
11. Maiden and Ncube (1998): Acquiring COTS Software Selection Requirements. IEEE Software March/ April 1998, pp. 46-56.
12. Nielsen, Jakob: The usability engineering life cycle. IEEE Computer, March 1992.
13. Nurmuliani, N., Zowghi, Didar and Fowell, Sue: Analysis of requirements volatility during software development life cycle (2004), IEEE. https://opus.lib.uts.edu.au/bitstream/10453/2603/3/2004001816.pdf
14. Redish, J., Molich, R., Bias, R. G., Dumas, J., Bailey, R., Spool, J. M.: Usability in Practice: Formative Usability Evaluations — Evolution and Revolution. CHI 2002, April 20-25, 2002, Minneapolis, USA.
15. Robertson, S. and Robertson, J. (2012): Mastering the requirements process.