

Task Descriptions versus Use Cases

Soren Lauesen, Mohammad A. Kuhail

IT University of Copenhagen, Denmark
slauesen@itu.dk, moak@itu.dk

Abstract. Use cases are widely used as a substantial part of requirements, also when little programming is expected (COTS-based systems, Commercial-Off-The-Shelf). Are use cases effective as requirements? To answer this question, we invited professionals and researchers to specify requirements for the same project: *Acquire a new system to support a hotline*. Among the 15 replies, eight used traditional use cases that specified a dialog between user and system. Seven used a related technique, task description, which specified the customer's needs without specifying a dialog. It also allowed the analyst to specify problem requirements - problems to be handled by the new system. It turned out that the traditional use cases covered the customer's needs poorly in areas where improvement was important but difficult. Use cases also restricted the solution space severely. Tasks didn't have these problems and allowed an easy comparison of solutions.

Keywords: use case; task description; software requirements; agile requirements; verification; COTS; interaction design; diffusion of innovation.

1 Background

Traditional requirements consist of a list of system-shall-do statements, but don't describe the context of use. IEEE-830, for instance, uses this approach [8]. The lack of context makes it hard for users to validate such requirements, and developers often misunderstand the needs (Kulak & Guiney [11]). Jacobson introduced use cases in the late 1980s [9]. They describe the dialog (interaction) between a system and a user as a sequence of steps. Use cases provided something that traditional requirements lacked, and they became widely used as a substantial part of requirements. Soon other authors improved on the basic idea and wrote textbooks for practitioners, e.g. Cockburn [5], Kulak & Guiney [11], Armour & Miller [3], Constantine & Lockwood [6].

Use cases have developed into many directions. Some authors stress that use cases should be easy to read for stakeholders and that they should not be decomposed into **tiny use cases** [5], [10]. The CREWS project claimed that use cases should be detailed and **program-like** with *If* and *While*, and have rules, exceptions and preconditions (reported in [1] and questioned in [7]). In contrast, Lilly [19] and Cockburn [5] advise against program-like elements. Other authors claim that use cases must have dialog details to help developers [26], [29]. In this study we saw examples of tiny use cases as well as program-like ones.

In the paper we will only discuss how use cases handle the user-system interaction. This is the most common use in literature as well as in practice.

Virtually nobody discusses whether use cases in practice are suited as verifiable requirements. Use cases seem to be intended for system parts to be built from scratch, but this situation is rare today. What if the use case dialog differs from the dialog in one of the potential systems? If we reject the system for this reason, we might reject an otherwise good system. If we accept it, what are the real requirements?

As a consultant, Lauesen had observed how use cases were used as a main part of requirements in large software acquisitions, even though the customer expected a COTS-based system with little add-on functionality. Usually the use cases were not used later in the project. However, in one large project the customer insisted on them being followed closely. As a result, the system became so cumbersome to use that the project was terminated [27].

Task Description is a related technique that claims to cover also the case where the system is not built from scratch. One difference is that tasks don't describe a dialog between user and system, but what user and system have to do together. The supplier defines the solution and the dialog - the customer shouldn't. Fig. 1 summarizes the differences. The task technique was developed in 1998 (Lauesen [12], [13]) and has matured since [14], [15], [17].

To get a solid comparison of the two approaches, we looked at a specific real-life project, invited professionals and researchers to specify the requirements with their preferred technique, and compared the replies.

2 The Hotline Case Study

The case study is an existing hotline (help desk). Hotline staff were not happy with their existing support system and wanted to improve the one they had or acquire a new one, probably COTS-based.

Lauesen interviewed the stakeholders, observed the existing support system in use, and wrote the findings in a three-page analysis report. Here is a brief summary: The hotline receives help requests from IT users. A request is first handled by a 1st level

Figure 1. Key differences between task descriptions and use cases

A task description	A use case
1. Describes what user and system do together. No dialog.	Describes a dialog: what user does and what system replies.
2. Allows problem requirements, e.g.: ABC is a problem, we want a solution.	Ignores problems where the analyst cannot describe a solution.
3. Requirement: Support the task and remedy the problems. Can be met to various degrees.	Requirement: Implement the dialog - even if bad? Or accept other dialogs?
4. Customer may give solution examples: what system might do. Supplier specifies his proposed solution.	Doesn't distinguish requirement from solution.
5. Free sequence of steps - almost.	Hard to describe a sequence-free dialog.
6. Suited for agile, waterfall and COTS-based.	Intended for development from scratch.

supporter, who in 80% of the cases can remedy the problem and close the case. He passes the remaining requests on to 2nd line supporters. A supporter has many choices, e.g. remedy the problem himself, ask for more information, add a note to the request, transfer it to a specialist, order components from another company, park the request, and combinations of these. The hotline is rather informal and supporters frequently change roles or attend to other duties. The report included a screenshot of the existing key screen: a list of pending support requests.

We invited professionals to write requirement specifications based on the analysis report. The invitation emphasized that we looked for many kinds of "use cases" and didn't care about non-functional requirements. It started this way:

We - the IT professionals - often write some kind of use cases. Our "use cases" may be quite different, e.g. UML-style, tasks, scenarios, or user stories. Which kind is best?

Participants could ask questions for clarification, but few did. The full analysis report is available in [16]. It started this way:

A company with around 1000 IT users has its own hotline (help desk). They are unhappy with their present open-source system for hotline support, and want to acquire a better one. They don't know whether to modify the system they have or buy a new one.

An analyst has interviewed the stakeholders and observed what actually goes on. You find his report below. Based on this, your task is to specify some of the requirements to the new system: use cases (or the like) and if necessary the data requirements.

We announced the case study in June-July 2009 to members of the Requirements Engineering Online Discussion Forum <re-online@it.uts.edu.au>, to members of the Danish Requirements Experience Group, and to personal contacts. The British Computer Society announced it in their July 2009 Requirements Newsletter. We got several comments saying: *this is a great idea, but I don't have the time to participate*. Lauesen wrote special requests to Alistair Cockburn and IBM's Rational group in Denmark, but got no reply.

We received 15 replies. Eight replies were based on use-cases and seven on tasks. Some replies contained separate data requirements, e.g. E/R models, and some contained use cases on a higher level, e.g. business flows. When identifying verifiable requirements, we looked at these parts too. The full replies are available in [16]. Here is a profile of the experts behind the replies:

Replies based on use cases. Decreasing requirement completeness:

- Expert A. A research group in Heidelberg, Germany. The team has industry experience and has taught their own version of use cases for 7 years.
- Expert B. Consultant in California, US. Has 15 years professional requirements experience. Rational-certified for 11 years.
- Expert C. A research group at Fraunhofer, Germany.
- Expert D. Researcher at the IT-University of Copenhagen. Learned use cases as part of his education in UK, but has no professional experience with them.
- Expert E. Consultant in Sweden with ten years experience. Specialist in requirements management. Have classes in that discipline.

Expert F. A software house in Delhi. Write use cases regularly for their clients. Were invited to write a reply on a contract basis, and were paid for 30 hours.

Expert G. Consultant in Sweden with four years experience in requirements.

Expert H. Consultant in Denmark with many years experience. Teaches requirement courses for the Danish IT association.

Replies based on task descriptions. Decreasing requirement completeness:

Expert I. Consultant in Sweden with many years experience. Uses tasks as well as use cases, depending on the project. Selected tasks for this project because they were most suitable and faster to write.

Expert J. Help desk manager. Has 15 years experience with programming, etc.

Expert K. A recently graduated student who has used tasks for 1.5 years. Little program experience.

Expert L. Researcher at the IT-University of Copenhagen. Has 25 years industry experience and has later used tasks for 10 years as a consultant and teacher.

Expert M. GUI designer with one year of professional requirements experience.

Expert N. Leading software developer with 12 years experience. Learned about tasks at a course and wrote the reply as part of a four-hour written exam.

Expert O. A research group in Bonn, Germany. The team has several years of industry experience, but little experience with the task principle.

3 Evaluation Method and Validity

We evaluated the replies according to many factors, but here we deal only with these:

- A. Completeness: Are all customer needs reflected in the requirements?
- B. Correctness: Does each requirement reflect a customer need? Some requirements are incorrect because they are too *restrictive* so that good solutions might be rejected. Other requirements are incorrect because they are *wrong*; they specify something the customer explicitly does not want.
- C. Understandability: Can stakeholders understand and use the requirements?

Validity threats: For space reasons we don't discuss all the validity threats here, but only the most important ones:

- Lauesen has invented one of the techniques to be evaluated. His evaluation of the replies might be biased. We have reduced this threat by getting consensus from several experts, as explained in the procedure below.
- The experts didn't have the same opportunities for talking to the client, as they would have in real life. True, but it seemed to have little effect. During the consensus procedure, there was no disagreement about which requirements were justified by the analysis report and which were not.
- Lauesen studied the domain and wrote the analysis report. This gave him an advantage. True, but it had no influence on the other six task-based replies. Further, when ranking the task-based replies on completeness (Fig. 6), Lauesen (Expert L) was only number 4. (His excuse for the low ranking is that he spent

only one hour writing the solution, plus 5 hours pretty-typing it without improving it in other ways.)

Procedure

1. The two authors have different backgrounds and independently produced two very different replies, Kuhail's based on use cases (Expert D) and Lauesen's on tasks (Expert L). We evaluated all replies independently. Each of us spent around 1 to 3 hours on each reply.
2. We compared and discussed until we had consensus. As an example, one of us might have found a missing requirement in reply X, but the other could point to where the requirement had been stated in the reply. We had around 5 points to discuss for each reply.
3. For each reply, we sent our joint evaluation to the experts asking for comments and for permission to publish their reply and our comments. We also asked for comments to our own solutions. Some authors pointed out a few mistakes in our evaluation, for instance that we had mentioned missing requirements in their reply that were not justified by the analysis report; or that our own solutions missed more requirements than we had noticed ourselves. We easily agreed on these points. Other authors said that our evaluation basically was correct, and that they were surprised to see the task approach, and considered using it in the future.
4. Finally we asked two supporters (stakeholders) to evaluate the four representative replies presented below. It was of course a blind evaluation. They had no idea about the authors.

We asked the supporters these questions in writing:

- a. *How easy is it to understand the requirements?*
- b. *Which requirements are covered [met] by the system you have today?*
- c. *Which requirements specify something you miss today?[want-to-have]*
- d. *Do you miss something in addition to what is specified in the requirements?*
- e. *Are some requirements wrong?*
- f. *Could you use the requirements for evaluating the COTS system you intend to purchase?*

The supporter got an introduction to the case and the style used in the first reply. He/she was asked to read it alone and answer the questions above. He/she spent between 30 and 50 minutes on each reply. Next we met, looked at the reply and asked questions for clarification. We handled the other replies in the same way.

Karin, senior supporter: The first supporter was Karin Tjoa Nielsen. Karin had been the main source of information when Lauesen wrote the analysis report. She is not a programmer but has a very good understanding of users' problems and the hot-line procedures. She read the replies in what we considered a sequence of increasing difficulty: Expert H (tiny use cases), Expert L (Lauesen, tasks), Expert A (program-like use cases), Expert K (tasks).

Morten, supporter with programming background: The second supporter was Morten Sværke Andersen. Morten is a web-programmer, but had worked in the hot-line until two years ago. He had never seen use cases, but had worked with user stories. He read the replies in this sequence: Expert L (Lauesen, tasks), Expert H (tiny use cases), Expert A (program-like use cases), Expert K (tasks). We chose this sequence to give Lauesen the disadvantage of being first.

4 Sample Replies and Stakeholder Assessment

The replies varied widely in time spent and length, but tasks tended to be faster to write and shorter than use cases. The lengths below cover only the use-case parts, not data descriptions, introduction, high-level flows, etc. Space characters are not included in the counts.

	Use cases	Task descriptions
Time spent	2.5 - 60 hours	2 - 6 hours
Length of use cases/tasks	5,500 - 37,000 characters	1,200 - 9,000 characters

Some use case replies were extremely hard to understand (B, F, G) and it wouldn't be fair to use them in the detailed comparison of use case principles against task principles. We will illustrate the replies with the four examples summarized in Fig. 2. We chose them because they are short, well-written examples of tiny use cases, program-like use cases, and task descriptions.

Task closure. One of the factors we will look at is *task closure*. A use case or task is closed if it covers what a user does without essential interruptions from start (trigger) to end (done for now). This is important because we want to make sure that the system supports the user well from start to end. Use cases may or may not be closed, and in this study many were not. They were either too small or too large. Tasks should be closed according to the guidelines.

Figure 2. Overview of four replies

Expert H (tiny use cases)	Expert A (program-like use cases)	Expert L (Lauesen) (tasks - no dialog)	Expert K (tasks - no dialog)
UC1. Record new request (IT user or supporter)	UC1. Trigger and control hotline problem solution (IT user)	T1: Report a problem (IT user)	T1: Report problem and follow up (IT user)
UC2. Follow up on request (IT user or supporter)	UC2. Accept request (supporter)	T2: Follow up on a problem (IT user)	
UC3. Add request data (IT user or supporter)	UC3. Clarify request (supporter, IT user, new supporter)	T3: Handle a request in first line (supporter)	T2.1: First line, handle request (supporter)
UC4. Transfer request (supporter)	UC4. Handle request (supporter)	T4: Handle a request in second line (supporter)	T2.2: Second line, handle request (supporter)
UC5. Update request (supporter)	UC5. Set support level (supporter)	T5: Change role (supporter)	T2.3: Change state - both lines (supporter)
UC6. Retrieve statistics (manager)	UC6. Get statistics (manager)	T6: Study performance (manager)	
UC7. Generate reminder (system use case)	UC7. Warn about orphaned requests (system use case)	T7: Handle message from an external supplier (supporter)	
		T8: Update basic data (manager)	
Closure: Use cases are too tiny for task closure.	Closure: A use case may extend over a long time and several users.	Closed tasks. One user.	Closed tasks. One user.
Problems covered: 2	Problems covered: 5.5	Problems covered: 7	Problems covered: 7.5
Length: 5500 chars. Time: 2.5 hours.	Length: 9900 chars. Time: Unknown.	Length: 4600 chars. Time: 6 hours.	Length: 2900 chars. Time: 3 hours.

A. Tiny use cases

Expert H's reply consists of seven use cases. Fig. 3 shows one of them in detail (*Transfer request*). It describes the dialog when a supporter wants to transfer a help request to another supporter. The steps alternate between *The system does* and *The user does*. Deviations from this sequence are recorded as variants below the main flow, e.g. the variant that the user wants to filter the requests to see only his own.

Expert H's use cases have a simple flow with few deviations from the main flow. They are examples of *tiny use cases*, where each use case describes a simple action carried out by the user. Real-life user activities would often include several of these use cases before task closure.

The reply is easy to read. However, it gives an inconvenient dialog if the system is implemented as described. As an example, several use cases start with the same steps (UC 1, 2 and 3). If implemented this way, the supporter will have to select and open the request twice in order to add a note to the request and then transfer it (a common combination in a hotline).

Supporter assessment: The supporters were confused about the supplementary fields with goal, actor, etc. and ignored them as unimportant. Apart from this, they found the reply easy to read, but concluded that it contained only few and trivial requirements. As an example, Morten considered the entire use case in Fig. 3 one trivial

Figure 3. Expert H: Dialog steps in one column. Tiny use cases.

USE CASE #04	Transfer request
Goal	Transfer a request to a specific hotline employee
Level	User goal (sea level)
Precondition	User is logged in and has the right to transfer requests
Postcondition	N/A
Primary, Secondary actors	Hotline
Trigger	Primary actor

NORMAL FLOW	
Step	Action
1	The system shows a list of all open requests
2	The user selects a support request
3	The system shows request data for the selected request
4	The system shows a list of hotline employees
5	The user chooses a hotline employee from the list
6	The system changes Owner to the selected employee and changes state to "2nd level"
7	The system updates the request

VARIANTS	
Step	Action
2a	User wants to filter his own requests
1	The system shows a list of all open requests with the user as Owner
2	The use case goes to step 2

requirement. *It is more of a build-specification*, he said. They noticed several missing or wrong requirements.

B. Program-like use cases

Expert A's reply also consists of seven use cases. Fig. 4 shows one of them in detail (*Handle request*). It describes the dialog when a supporter takes on a help request and either corrects the problem or transfers the request to someone else. The steps are shown as two columns, one for the user actions (A1, A2, etc.) and one for the system actions (S1, S2, etc.). Variants are shown right after the related step. There are many variants, if-statements, included use cases, rules and exceptions. Special notation is used to show whether a step is optional, and whether it terminates the use case if done. The description is a kind of program that specifies the possible sequences.

The optional steps may be carried out in any order, which gives the user much freedom to choose the sequence.

Some of A's use cases involve several users and describe a kind of dialog between them. As an example, use case *Clarify Request* describes that a supporter can require more information from the IT user, what the user does, and how a (new) supporter handles it. These use cases are too large to match a closed user task.

Expert A's use cases are very different from H's. As an example, H's entire use case 4 (Transfer request) is just step A3 in A's *Handle request*. An H use case shows a tiny part of the dialog, while an A use case covers a more coherent period. This is not just a matter of formatting, but a different approach to the modeling.

Supporter assessment: Karin (the senior stakeholder) couldn't understand the program-like details and ignored them. She found the rest okay to read and used it as a checklist. She identified many lines as requirements that were *met* or *want-to-have*. She also noticed some missing or wrong requirements. Morten (the programmer) found the reply very hard to read and spent a lot of time checking the program-like parts. *If I had been asked to read this first, I wouldn't have done it*, he said. He found many parts wrong or dubious. He concluded that the use cases were useless for checking against a new system, since it might well work in some other way.

Figure 4. Expert A: Dialog steps in two columns. Program-like use cases.

Name	Handle request	
Actor	Supporter (first/second line)	
Supporting Actors	IT user	
Goal	Solve a problem.	
Precondition	[Workspace: request]	
Description	Actor	System
	A1) VAR1) the actor takes on an open request from his/her line. [Exception: No open requests] VAR2) the actor receives a request forwarded to him/her.	S1) If VAR1) The system records the actor as owner of the request. [System function: take on request]
	A2) [optional *] The actor adds information [Include UC Clarify request]	
	A3) [optional X] The actor forwards the request. VAR1) forward to second line VAR2) forward to specific expert (no matter what line) [include UC Handle request]	S3) The system forwards the request [System function: forward request] If VAR1) The system changes the owner to "not set" and the status to "second line" If VAR2: The system sets the expert as the owner and notifies him/her about the forwarded request. If he/she is logged in at the moment, the system sends an alert in a way designed to attract his/her attention (e.g. a pop-up window). Else it sends the alert as soon as the expert logs in.
	A4) [optional *] The actor looks up information of the request.	S4) The system provides information about the request. [System function: show request details]
A5) [optional X] The actor solves the problem and closes the request.	S5) The system closes the request. The system sends the user a notification. [System function: close request]	
Exceptions	[There are no open requests]: The system contains no open requests.	
Rules	None	
Quality Requirements	None	
Data, Functions	System functions: take on request, show request details, add information to description field . . .	
Post conditions	The request is closed in the system.	
Included UCs	Clarify request; Handle request	

C. Task descriptions (Lauesen)

Expert L's reply consists of eight task descriptions. Fig. 5 shows one of them in detail (Handle a request in second line). It describes what a second-line supporter can do about a request from the moment he looks at it and until he cannot do more about it right now (a *closed task*). He has many options, e.g. contact the user for more information, move to the problem location, order something from an external supplier - or combinations of these.

At first glance, task descriptions look like use cases, but there are several significant differences:

1. The task steps in the left-hand column specify what user and computer do together without specifying who does what. Thus no dialog is specified. In this way you avoid inventing requirements about what you believe the system must do. Variants of the task step are shown right after the step, e.g. *1a* about being

Figure 5. Expert L (Lauesen): Tasks - no dialog. Problems as requirements.

C4. Handle a request in second line

Start: The supporter gets an email about a request or looks for pending requests.

End: The supporter cannot do more about the request right now.

Subtasks and variants:		Example solutions:
1	Look at open second-line requests from time to time, or when finished doing something else.	
1p	Problem: In busy periods it is hard to spot the important and urgent requests.	Can restrict the list to relevant requests. Can sort according to reminder time, priority, etc.
1a	Receive email notification about a new request	
2	Maybe contact the user or receiver to obtain more information.	p, q: Problems today a, b: Variants of the subtask
3	Maybe solve the problem by moving to the problem location.	
4	Maybe work for some time on the problem. Inform others that they don't have to look at it.	Put the request in state <i>taken</i> .
5	Maybe order something from an external supplier and park the request.	The system warns if no reminder time has been set.
6	In case of a reminder, contact the supplier and set a new reminder time.	User and computer together
6p	Problem: The user doesn't know about the delay.	The system sends a mail when the reminder time is changed.
7	Maybe close the case.	The system warns if the cause hasn't been set.
7p	Problem: To gather statistics, a cause should be specified, but this is difficult and cumbersome today.	Maybe: The user decides
7q	Problem: The user isn't informed when the request is closed.	The system sends a mail when the request is closed. The supporter has the possibility to write an explanation in the mail.
8	Maybe leave the request in the "in-basket" or transfer it to someone else.	

- notified by email.
2. You can specify problems in the way things are done today, e.g. *1p* about spotting the important requests. You don't have to specify a solution.
 3. The requirements are that the system must support the tasks and remedy the problems as far as possible. You can compare systems by assessing how well they do this.
 4. In the right-hand column you may initially write examples of solutions, later notes on how a potential system supports the step. Developers (also agile ones) report that the left-hand side is relatively stable, but the right-hand side is not. Sometimes the system can carry out the entire step alone, for instance if it automatically records the IT user's name and email based on the phone number he calls from.
 5. The steps may be carried out in almost any order. Most of them are optional and often repeatable. The user decides what to do and in which sequence. The steps are numbered for reference purposes only. (Cockburn and others also recommend this, but it is hard to realize with use cases, because a dialog by nature is a step-by-step sequence.)

Problem requirements: In fig. 5 there are four problems in the way things are done today. Some of them, for instance 7p, are very important but not easy to solve.

When a step has an example solution, the solution is not a requirement. Other solutions are possible. Steps without a solution may have an obvious one that the analyst didn't care to write (e.g. step 6) - or he cannot imagine a solution (e.g. 7p). Both are okay.

Lauesen's task list is quite different from expert H's use case list. As an example, H's use case 3, 4 and 5 are steps in Lauesen's *Handle a Request in Second Line*. The relationship to A's use cases is more complex. An A use case may go across several users and a large time span. In contrast a task should be closed, i.e. cover what one user does without essential interruptions from Start (trigger) to End (done for now).

Supporter assessment: The senior supporter was excited about this reply: *This is so clear and reflects our situation so well. It is much easier to read than the first one I got [the tiny use cases]. I hope I don't step on someone's toes by saying this.* Although she correctly understood the left-hand side, she tended to consider the example solution a promise for how to do it. Morten read this reply as the first one. He was excited about the start and end clauses because they reflected the real work. He was puzzled about the distinction between first and second line, because the hotline didn't operate in that way when he worked there. Initially he hadn't marked the problems as requirements, but later included them. Both supporters marked most of the steps as requirements that were *met* or *want-to-have*. They didn't notice any missing or wrong requirements.

We have seen also in other cases that the two-column principle and the problem requirements are not fully intuitive. However, once reminded of the principle, readers understand the requirements correctly. (The reply started with a six-line explanation of the principles, but the supporters didn't notice it.)

The senior supporter decided to evaluate the system they intended to buy by means of the task descriptions (she still didn't know whom the author was). She did this on her own a few days later, and concluded that most requirements were met - also those

that earlier were *want-to-have*. She could also explain the way the new system solved these *want-to-have* problems.

Task description: Expert K's reply consists of four tasks, very similar to Lauesen's, but K hasn't covered statistics and maintenance of basic data. For space reasons we don't show a detailed task. In general the task replies are rather similar because they follow the closure principle. We chose K's reply because K had no hotline domain experience, but had some task experience from other domains.

Supporter assessment: The senior supporter read this reply a few months after reading the first three replies. She found also this reply easy to read. She had marked eight of the requirements with a smiley and explained that these requirements showed that the author had found the sore spots in a hotline. (Seven of these requirements were problem requirements.) She had marked all the requirements as *met* in the new system, but explained that they currently worked on improving the solution to one of the problems (quick recording of problems solved on the spot). She had noticed that requirements for statistics were missing.

Morten also read this a few months later. He was asked to compare the reply against the old system, since he didn't know the new one. He found the tasks easy to read and noticed six requirements that were not met in the old system (five of them problem requirements). He didn't notice that requirements for statistics were missing.

He also made comments that showed that the task principles had to be reinforced. He surprisingly suggested that the step *estimate solution time* should be deleted - otherwise supporters would be annoyed at having to make an estimate. He forgot that all task steps are optional, and you don't have to carry out an optional step. In a few places he complained about missing or bad *solutions*. He forgot that the solution side is just examples. The supplier should specify the real solution.

5 Completeness - Dealing with Existing Problems

Requirements are complete when they cover all the customer's needs. Completeness of the ordinary requirements varied within both groups of replies due to participant's experience, time spent, etc. However, the means were almost the same:

- Number of ordinary requirements covered by a use case reply: 18.3 ± 3.2
- Number of ordinary requirements covered by a task reply: 17.6 ± 4.8

However, the two groups handled the present problems very differently. If requirements don't cover these problems, the customer may end up with a new system that doesn't remedy the problems. He may not even notice that the problems remain because they don't call his attention while he verifies the requirements, e.g. during acceptance testing.

The analysis report mentions nine problems in the existing hotline, e.g.: *In busy periods, around 100 requests may be open (unresolved). Then it is hard for the individual supporter to survey the problems he is working on and see which problems are most urgent.*

In principle requirements can deal with such a problem in three ways:

1. Specify a solution to the problem.
2. Ignore the problem.
3. Specify the problem and require a solution (a *problem requirement*).

Problem requirements are unusual in traditional requirements, but are used extensively in tasks. The use case replies deal with problems by either specifying a solution or ignoring the problem.

All the use case replies ignored the problem with the busy periods. Expert A confirmed our suspicion that it was because they couldn't see a solution.

Expert A explained: *We do not think that this can be solved by the system. The system gives all the important information (e.g. date placed) for the user to decide.*

With tasks you can just record the problem, for instance as in Fig. 5, 1p. Lauesen's experience from many kinds of projects is that recording the problem helps finding a solution later, for instance the one outlined in Fig. 5, 1p. Even if the analyst cannot imagine a solution, a supplier may have one. As an example, we considered problem 7p (specifying the cause of requests) hard to solve until we saw a product where the supporter could choose from a short experience-based list of the most common causes, and a longer tree-structured list for the remaining causes. The short list covered around 90% of the help requests. The customer could edit and add to the lists, of course.

The analysis report mentions nine problems. For each of the 17 replies we checked which problems it covered, either as a problem requirement or as a solution. Fig. 6 shows the results. The problem above is recorded as problem A. None of the use case replies deal with it. Problems that have an easy solution, for instance problem I, are covered by most replies.

Tasks cover these nine problems significantly better than use-cases do:

- Number of problems covered by a use case reply: 3.8 ± 1.2
- Number of problems covered by a task reply: 6.5 ± 1.9

The difference is significant on the 1% level for ANOVA ($p=0.6\%$) as well as for a t-test with unequal variances ($p=0.5\%$).

The standard deviations are due to differences in expertise, time spent, simple mistakes, etc. Not surprisingly, the deviation for task replies is larger because some task participants had little task experience (N and O).

Maiden & Ncube [21] observed that when comparing COTS products, all products meet the trivial requirements. Selection must be based on the more unusual requirements. In the hotline case, the problem requirements are the ones that will make a difference to the customer, while the ordinary requirements will be met by most systems. As a result, use cases would not help, but tasks would.

Figure 6. Problem coverage: 1 = fully covered, 0.5 = partly covered.

Use-case based:	A: Hard to spot important requests	B: Difficult to specify a cause. May change later	C: User: When can I expect a reply?	D: Forgets to transfer requests when leaving	E: Cumbersome to record on-the-spot solutions	F: User: When is it done?	G: Nobody left on 1st line	H: Reminder: Warn about overdue requests	I: Today it is hard to record additional comments	Total
Expert A				1	1	1	1	0.5	1	5.5
Expert B		0.5	1		1	1		0.5	1	5
Expert C				0.5		1	1	0.5	1	4
Expert D				1			1	1	1	4
Expert E					1		1	1	1	4
Expert F						0.5	1	1	1	3.5
Expert G			0.5					0.5	1	2
Expert H						0.5		0.5	1	2
Total UC	0	0.5	1.5	2.5	3	4	5	5.5	8	

Task-based:										
Expert I	1	1	1	1	1	1	1	0.5	1	8.5
Expert J	1	1	1	1	1	1	1		1	8
Expert K	1	0.5	1	1	1	1	1	1		7.5
Expert L	1	1	1	1	1	1		1		7
Expert M	1	1	0.5		0.5	1	1		1	6
Expert N	1	1			0.5	1			1	4.5
Expert O					0.5	1	1		1	3.5
Total tasks	6	5.5	4.5	4	5.5	7	5	2.5	5	

6 Compare Systems - Too Restrictive Requirements

In the hotline case the purpose is not to develop a new system, but to expand the existing one or acquire a new one (probably COTS-based). With the task technique stakeholders check how well the system supports each task step and each problem. In order to compare systems, they check each of them in the same way.

We couldn't see how the use cases handled this situation. The use cases specify a dialog in more or less detail, and it seems meaningless to compare this dialog with a different dialog used by an existing system. The requirements arbitrarily restrict the solution space. So we asked participants how their use cases could be used for comparison with an existing system. Here are some of the replies:

Expert F: *We will make use of something that we call a decision matrix. We have prepared a sample for your reference where we are comparing the present Hotline system with the system that we propose...*

Their matrix shows nine features to compare, e.g. *Reminder management* and *Automatic request state management*. They have no direct relation to the use cases. F explained that their use cases specified a new system to be developed from scratch.

Expert H: *Use Cases are an optimal source for defining Test Cases. So running these Test Cases against different proposals and the existing system, it will be clear which systems fulfill most of the functional requirements. The tests can be run "on paper" since many solutions have not been developed yet.*

This seems a good idea. The only problem is at what detail you define these test cases. Assume that you use the use cases directly as test scripts. You would then test *Transfer request* (Fig. 3) by trying to select a request from the list, checking that the system shows request details and later shows a list of hotline employees. But what about a system where you don't need to see the request details, but can transfer the request directly from the list? You would conclude that the system doesn't meet the requirements (actually it might be more convenient). One of the supporters (Morten) actually tried to verify Expert A's program-like use cases in this way, but became so confused that he gave up.

Hopefully, the testers have domain insight so that they can abstract from the details of the use case and make the right conclusion. If so, most of this use case is superfluous. You could omit everything except the heading *Transfer request* and simply test that the system can transfer a request, and make a note about how easy it is to do so. The other supporter (Karin) actually verified all use cases in this way.

Expert A: *Our requirements only describe the to-be system. They cannot be used to compare the other systems with the current (open source) system. However, they could be used to see whether the supplier's system (if not available, the description of the system) meets our requirements.*

Expert A (another team member): *I think the purpose of your [Lauesen's] specification is quite different from ours. We want to provide a specification that describes the*

solution to the problems on a high level. For the purpose of choosing between solutions your specification is much better, but this was not so prominent in the experiment description that we considered it the major context.

The other use-case replies follow the same lines. They describe a future solution in detail.

7 Wrong requirements

The supporters and we noted several wrong requirements in the use cases. As an example, Expert C mentions these two business rules:

R1. Only problems with high priority may be requested via phone or in person.

R2. For statistical purpose it is not allowed to create a request for more than one problem.

None of these rules are justified in the analysis report, and it would be harmful to enforce them. Should hotline reject a user request if it contains more than one problem? The hotline would surely get a bad reputation.

We believe that use case theory and templates cause these mistakes. Many textbooks on use cases emphasize rules, preconditions, etc. and their templates provide fields for it. Most replies used such a template, and as a result, the authors were tempted to invent some rules, etc. Often these rules were unnecessary or even wrong.

In order to avoid this temptation, tasks do not have fields for preconditions or rules. When such a rule is necessary to deal with a customer need, it can be specified as a task step (e.g. *check that the request has a high priority*), as a constraint in the data model (e.g. *in the closed state, a request must have a cause*), or in other sections of the requirements [15].

8 Discussion

When we presented the first, short version of this paper at the REFSQ'11 conference, it caused an unusually long debate. In this section we will discuss some of the issues brought up.

A. Addressing the wrong acquisition context

Why did all the use case authors describe a future solution and later realize that it wasn't useful as requirements in this case? The analysis report said that the customer wanted to modify the existing system or buy a new one - not build a new one. They addressed the wrong acquisition context.

Lauesen has seen this problem over and over, also in very large acquisitions. The analysts define detailed use cases and other requirements although they know that most parts of the system will be COTS. Even if they expect the system to be built from scratch, strict adherence to the use cases will guarantee a poor user dialog. We believe that use case principles and current practice are the causes:

1. Use case principles force you to design a dialog at a very early stage. In this way you design key parts of the solution rather than specifying the customer's needs.
2. Use cases are so widely used that nobody questions their usefulness.
3. Few analysts know alternative requirements that specify the user-system interaction without specifying the dialog.

The classical requirements textbooks [e.g. 30 and 24] don't mention COTS at all. The modern ones say a bit, but not enough to evaluate a COTS-based system's user-computer interaction. Wiegers suggest that *use cases* work well [28, p289] and Alexander & Beus-Dukic [2] suggest a comparison of COTS-based solutions based on higher-level criteria similar to expert F's decision matrix above.

B. How use cases and tasks deal with the step sequence

Although Cockburn and others emphasize that the step sequence should be rather free, it causes problems in practice. First of all, why do you have to describe a sequence at all? This seems necessary because you describe a dialog of the form: *the user does - the system does*. You are all the time encouraged to specify what happens next.

The habit might also come from the simple examples used in textbooks, for instance the ATM example. Here a strict sequence is okay. Analysts believe that they should do something similar in the complex cases they deal with.

One way to avoid a strict sequence is to use tiny use cases, each of which performs one simple action. Since there is no prescribed sequence between use cases, this allows the user to choose his own sequence. However, we don't believe analysts make tiny use cases for this reason.

The consequence of tiny use cases is that you generate a lot of useless formalities (preconditions, primary user, exceptions, etc.) and invent dialog steps to prepare the essential action (e.g. the first steps of *transfer request*). Yet, the true context of use isn't visible. Fig. 3 is a good example.

Another way is to "program" the dialog and its different flows. Many authors describe a main flow and alternate flows. Each of these has a sequence of steps. Other authors use exceptions, if-statements, variants, etc. to describe the possible sequences. This makes the use cases hard to read and introduces many unnecessary requirements about sequence and rules. Fig. 4 is an example.

Some analysts suggest using more precise specification languages with parallelism, such as UML activity diagrams. This doesn't help because the basic problem is that the dialog shouldn't be specified in the requirements. Furthermore, such specifications would make the requirements harder to understand for the real stakeholders.

HCI specialists have tried to model tasks for many years, focusing on what users actually do with an existing system. It was hard. There were too many variations and special situations to deal with, and different users carried out the same task differently. They did not seem to follow a procedure in the computer-sense of the word [22, 20]. HCI specialists seem to have concluded that precise task modeling is unrealistic for non-trivial tasks [23].

Task descriptions avoid the sequence problem by not specifying a sequence. Further, all subtasks are in principle optional. The user decides which subtasks to carry out and in which sequence. There may be preconditions for a task step, e.g. that the supporter must have selected a request before he can transfer it. However, this goes without saying in the requirements. We can leave it to the programmer.

Example: We will illustrate the sequence problem with an example everybody knows: the dialog in a popular system such as MS Word. Imagine that users were forced to follow this logical flow:

Create headings, create body text, create figures ...

Authors would hate it unless we also supported a lot of alternate flows so that authors might specify some body text first, then a heading.

As an alternative, we might define a lot of tiny use cases, for instance: Create heading, Create paragraph, Edit heading, etc. This would allow the author to choose the sequence he likes. However, such a use case is too small to meet a meaningful goal and doesn't reflect the true work situation. It might result in a system where the author had to select what to do, for instance *Edit Heading*, next select the heading, etc. We might not even get a full view of the text because the need for such an overview isn't visible in the requirements.

Using tasks, we would specify one task with a free step sequence:

Task 1: *Edit document.*

Start: The user has time for working on the document.

End: No more editing to do right now.

1. *Get an overview of the document and read parts of it.*
2. *Maybe see what was changed recently.*
3. *Maybe create or edit a heading.*
4. *Maybe create or edit a paragraph ...*

...

20. *Park the document for later editing.*

The hotline case is actually closer to this than to an ATM. The supporter finds a request to deal with and opens it to see the details. Then he chooses what to do, for instance add a note, change priority, transfer the request, park the request for now. He can do this in almost any sequence. Attempts to prescribe a sequence will make his job more difficult.

Task descriptions versus traditional shall-statements: A task description looks suspiciously like a list of traditional functional requirements such as these:

The system shall have these functions:

*Create a heading,
Edit a heading ...*

Why not write such a list rather than a task? The important difference is that a task lists the functions needed in a specific closed use context. You can see how the functions cooperate, and you can check that the functions are convenient to use in this context. Tasks are not a grouping of functions because a function may be used in several tasks. In the hotline case, a function such as transfer request may be used in first-line tasks as well as second-line.

IEEE 830 [8] shows grouping of functions according to various criteria, e.g. by object or by user class, but doesn't suggest grouping according to closed use contexts. (And if done, it wouldn't be a grouping anymore).

Traditional requirements supplement the *shall* requirements with various ways of describing the context, for instance *context diagrams* [28, 30, 2] and *rich pictures* [4,

2] or *operational requirements* such as *the product will be used at freezing temperatures and the users will wear gloves* [24]. These are definitely useful, but not sufficient to serve as requirements for the user-system interaction.

C. What comes after the tasks when you develop a solution?

At the conference, Martin Glinz and others asked: If we are going to develop a new system, what comes after task description?

When you use tasks, you don't describe a dialog. So it seems tempting to design the dialog next, for instance as use cases or some flow diagram. Many web-designers do so and next design a screen for each step in the dialog. The result is the cumbersome dialogs we meet on many web-sites. Most of us prefer "one-click shopping".

Experience from many projects suggests that it is better to design the data-carrying screens first, i.e. the screens that show the important data the user needs to see. We call these screens *Virtual Windows* [14]. They show detailed data in a realistic layout, but they don't contain functions, such as *Save* buttons, menus or links to another screen. In the hotline case, one of the Virtual Windows could show all the details of a single request. Another virtual window could show a list of pending requests. Since Virtual Windows don't have functions, we cannot use them to "operate" the system. They are not the same as the mockup screens used by interaction designers for usability testing [23].

It is important to support each task with as few Virtual Windows as possible in order to reduce the mental load on the user. It is also important that we look at the entire closed task, i.e. what the user does from trigger to task closure without essential interruptions.

The Virtual Window method has additional guidelines: Reuse screens across tasks and task steps, strive for few screens in total, provide good overview of data, make a rather detailed graphical design of each screen, review the screens with typical users and improve the design until users are happy.

The next step is to add functions (buttons, menus, etc.) to each screen so that it can do something. Here you look at the task steps one by one, identify the screens needed, and identify the necessary functions to support the step. For the hotline system you would identify a function to transfer a request to someone else, a function for sending a reply to the IT user, etc. These functions could be buttons, combo-boxes or short-cut keys on the screen that shows the request details. A typical task step will use a function or two, but functions are reused across tasks so that the result is a modest number of functions in total. The user can use these functions in almost any sequence and in this way create his "own" dialog.

When all of the functions have been added, for instance on a mockup, it is possible to let typical users try to "operate" the system, while a developer simulates what the system would do, and a few other developers observe what goes on. This is usability testing - the only reliable way to find out why real users cannot operate the system without help from someone else. In order to make the system meet usability requirements, you have to revise the design a couple of times. The tasks come in handy to help select realistic test cases (test tasks).

The experience is that when you design the Virtual Windows first, it becomes easy to define the necessary functions to support a free user dialog. Amazingly, in the final

system, the tasks are not visible. They were just a scaffold for building a convenient user interface.

In the hotline case, four of the seven task replies include a user interface built on the Virtual Windows approach (replies K, L, N and O). The user interfaces are surprisingly similar and consist of three screens: (1) A list of requests somewhat similar to the one in the analysis report. (2) A single request with all details. (3) A list of supporters with their current state. One of the use-case replies also included a user interface (reply A). It had a few more screens customized for specific use-case steps.

Lauesen's web-site [18] has examples of larger user interfaces designed with Virtual Windows. These examples also show the task descriptions that are the base of the design. One example is a redesign of Facebook for mobile use. It reduced the number of web-pages from around 40 to around 10, and even added some functionality that users missed.

D. How tasks were "invented"

Until 1997 Lauesen thought that it was unrealistic to describe larger tasks in any detail, and HCI specialists later seemed to conclude the same [23].

In 1997, Lauesen saw Cockburn's template for use cases [5] with real-life examples, and suddenly realized that task descriptions were possible and that they even scaled up to large systems. However, which of the many kinds of use case were best in practice?

Together with Marianne Mathiassen (Masters student at that time) and a team of analysts at a Danish Hospital, Lauesen tried to answer these questions: Should we describe what the user does? What the computer does? What they do together? As-is? Or to-be?

Working on the most difficult part of the hospital's current software acquisition (roster planning), they tried out many combinations using one, two, three or even four columns. The theoretically correct combination had four columns: user as-is, system as-is, user to-be, system to-be. They assessed each combination in light of its usefulness in the acquisition process. There were several conclusions:

1. More than two columns were confusing, and the columns tended to repeat what the other columns said.
2. It was best to describe what user and computer did in combination. This allowed customers to compare with the suppliers' ways of doing things. Surprisingly, this was the only combination that Cockburn rejected. He insisted that it must be clearly described who did what. (As the case study shows, the result is a premature design of the dialog.)
3. It wasn't interesting to describe what users did today. The purpose of the new system was to make users work in a different way, so why describe the old way? With one exception: Things that were problematic today should be described in order to look for systems that remedied these problems.
4. It wasn't interesting to describe what the old system did, except where it was problematic. But it was highly interesting to describe what the new system would do. However, the supplier or developer should do this. The customer might describe a solution he imagined, but it shouldn't be a requirement, because this might lead to excessively expensive solutions.

When the acquisition process was over, Lauesen assessed the requirements with the three suppliers who had sent a proposal. This caused only minor adjustments of the approach. Overall the suppliers were happy with the approach, particularly because it allowed them to offer something that was cheaper and still met the customer's needs, and because it allowed them to show solutions that exceeded the customer's expectations [12, section 10.7].

For a few years, Lauesen believed that the task approach was only for acquisition of COTS-based products. Then it proved to be just as useful for ordinary development projects (also agile ones) and product development. The main strength was that the customer's demands could be caught in such a way that developers could see the problem and had space for inventing a good solution.

Until 2001, Lauesen considered task descriptions a variant of use cases and called them *use cases in task notation*. Soon readers and colleagues convinced him that there was a much more profound difference between the two techniques. Today we simply call them tasks - a term that the HCI community has used extensively for decades.

E. Diffusion of the task technique

At the conference, people asked why the technique wasn't used widely when it seemed to have many advantages. As an example, Björn Regnell reported that he had been teaching the task approach for years, but when he met students a couple of years later in industry, they had forgotten about it and wrote use cases, because *this is the way in our company*.

The short explanation of the lack of spreading is that our research ideals are an illusion. The ideal is that once you have published something, other researchers and practitioners can read it, and if the result is interesting it spreads by itself. Most of us know that things don't work this way. However, we hope our students will bring the idea into practice. Unfortunately, this is also hard as witnessed by Regnell.

Lauesen has had some success spreading the task principle, primarily in his home country, Denmark. This is done not only through teaching, but also by giving courses and seminars in industry (around eight a year), and by providing consultancy for specific projects. Consultancy has the largest impact, because it enables practitioners to use the technique in their own projects. It also helps the researcher improve his approach to deal with situations he didn't know about. Every now and then Lauesen gets into contact with small communities abroad who use task descriptions and ask for additional advice. However, the critical mass is missing.

In contrast, use cases have spread widely. What is the difference? Everett Rogers' theory about *diffusion of innovation* [25] can shed some light on it. Rogers talks about five driving factors. We will discuss each of them.

Relative Advantage: How improved is the innovation against the existing way?

As the paper shows, there are significant advantages of tasks. However, they are not immediately visible. People's first reaction when seeing a task description is: This looks like a use case. What is new? Although people can see and understand the advantage when seeing a real-life example, they cannot immediately see how it applies in their own projects.

When use cases spread in the nineties, there was no competing technique that could specify the context of use in a systematic way. Structured Analysis and Rich Pictures [30, 4] were common at that time, but too vague to deal with user-system interaction.

Compatibility: How easy is it to assimilate the innovation into the individual's present life?

It is very hard to introduce tasks in a developer culture that is accustomed to use cases. It is a pioneer's job, not an activity for the typical developer.

Could it be introduced through the big consultancy organizations? This was actually attempted. The head of the Rational Group in IBM Denmark had heard about the advantages of tasks. In March 2009 he contacted Lauesen to find a way to include tasks in Rational. Lauesen found it a great idea and cooperated for some weeks with Rational staff. Soon the parties agreed that it wasn't feasible. Too much had to be changed in the course material, in the Rational tool and in the consultants' training. Although Rational staff didn't say so explicitly, there wasn't sufficient new business in the idea. Tasks were not a competitor.

In contrast, use cases had the advantage of being introduced as part of the OOA/OOD wave.

Maybe tasks should be introduced together with the spreading agile development, as a replacement for the fuzzier *user stories*. Agile is a competitor to approaches such as Rational. Usually agile developers are very aggressive against requirements. *It is not possible to specify requirements up front. It is only a waste of time.* However, when they have seen a couple of tasks, they become fascinated. *Oh, if these are requirements, they might be useful.* Some agile teams have used task descriptions (with a bit of guidance from an expert), and reported back that they are very useful. *The left-hand column is very stable, but we change the solution column a lot.*

Simplicity: How easy is it to use for the individual?

Tasks seem easy, but when analysts try on their own, 90% of them write traditional use cases in the new template - or traditional requirements in column 1 of the template. This is all wrong - column 1 should describe what user and computer do together. After feedback from a task expert, most of them succeed.

We don't think this difficulty is much different from use cases. In the case study, we saw several lengthy and confusing use cases from seasoned practitioners.

Trialability: How easy is it to experiment with the innovation?

You can easily experiment with writing task descriptions, but it is harder to try them in practice. The real feedback is during development, but analysts can get the first feedback when reviewing the tasks with stakeholders. This is often very encouraging, because users can see how tasks relate to their work situation. Karin's comments on the task descriptions in section 4 are a good example.

As this case study shows, use cases are much weaker in this area. Stakeholders have trouble understanding them or find them trivial. Some seasoned developers told us that the purpose of use cases isn't to communicate with stakeholders, but to impress them: *When stakeholders ask about progress, we give them 1000 pages of [tiny] use cases. This makes them shut up.*

Observability: How visible is the innovation to others?

In a specific project, tasks are visible to other developers and to stakeholders. When successful, they bring them into their next project. This is actually the way tasks spread today. But it is not the same visibility as if university teachers and large consultancy companies started using the technique.

Use cases are already very visible on the market, carried by consultants and university teachers.

9 Conclusion

In this study we compare real-life use cases against the related technique, task description. We deal only with use cases that specify the interaction between a human user and the system. We do not claim that the findings can be generalized to other kinds of use cases, for instance system-to-system use cases.

The study shows that with use cases, the customer's present problems disappear unless the analyst can see a solution to the problem. The consequence is that when the customer looks for a new system, he will not take into account how well the new system deals with the problems. Even if the analyst has specified a solution, a better solution may not get the merit it deserves because the corresponding problem isn't visible in the use cases.

Task descriptions avoid this by allowing the analyst to state a problem as one of the "steps", with the implicit requirement that a solution is wanted (*a problem requirement*). Example solutions may be stated, but they are just examples - not requirements. In practice, stakeholders need some guidance to understand these principles.

The study also shows that use cases in practice produce too restrictive requirements for two reasons: (1) They force the analyst to design a dialog at a very early stage, in this way designing a solution rather than specifying the needs. Often the dialog would be very inconvenient if implemented as described. (2) Many use case templates provide fields for rules, preconditions, etc. and these fields encourage analysts to invent rules, etc. Often the rules don't reflect a customer need and may even be harmful.

Task descriptions don't specify a dialog but only what user and system need to do together. The supplier defines the solution and the dialog, and stakeholders can compare the solution against the task steps to be supported. Tasks don't tempt the analyst with fields for rules, etc. When rules are needed, the analyst must specify them as separate task steps or in other sections of the requirements.

Tasks are also a good basis for designing the user interface because the developer can focus on designing screens that conveniently show the data needed during the task. He can add functionality and the dialog later.

Unfortunately, the task method doesn't spread easily. Use cases had the advantage of spreading with OOA/OOD and powerful consultants. Using tasks instead, requires a lot of change in present practice and tools.

References

1. Achour, C. B., Rolland C., Maiden N. A. M., Souveyet, C.: Guiding Use Case Authoring: Results of an Empirical Study. Proceedings of the 4th IEEE International Symposium (1999)
2. Alexander, I., Beus-Dukic, L.: Discovering requirements. Wiley, 2009.
3. Armour, F., Miller, G.: Advanced Use Case Modeling, Addison-Wesley (2001)
4. Checkland, P.B.: Systems Thinking, Systems Practice. John Wiley & Sons, Chichester, 1981.
5. Cockburn, A.: Writing Effective Use Cases, Addison-Wesley (1997 and 2000)
6. Constantine, L. L., Lockwood, L. A. D: Software for Use: A practical guide to the Models and Methods of Usage-Centered Design, Addison-Wesley, New York (1999)
7. Cox, K., Phalp, K.: "Replicating the CREWS Use Case Authoring Guidelines", Empirical Software Engineering Journal, Vol. 5, No. 3, pp. 245-268 (2000)
8. IEEE Recommended Practice for Software Requirements Specification, ANSI/IEEE Std. 830 (1998)
9. Jacobson, I., Christerson, M., Johnsson, P., Övergaard, G.: Object-Oriented Software Engineering - a use case driven approach. Addison-Wesley (1992)
10. Jacobson, I.: Use Cases: Yesterday, Today, and Tomorrow, IBM Technical Library (2003)
11. Kulak, D., Guiney, E.: Use Cases: Requirements in Context, Addison-Wesley (2000)
12. Lauesen, S.: Software requirements - styles and techniques. Addison-Wesley (2002)
13. Lauesen, S.: Task Descriptions as Functional Requirements. IEEE Software, March/April, pp. 58-65 (2003)
14. Lauesen, S.: User interface design - a software engineering perspective. Addison-Wesley (2005)
15. Lauesen, S.: Guide to Requirements SL-07 - Template with Examples, 2011, ISBN: 978-87-992344-1-7. Also on: www.itu.dk/people/slauesen/SorenReqs.html#SL-07.
16. Lauesen, S., Kuhail, M. A.: The use case experiment and the replies (2009): www.itu.dk/people/slauesen/
17. Lauesen, S.,Kuhail, M.: Use cases versus task descriptions. In: D. Berry and X. Franch (Eds.): REFSQ 2011, LNCS 6606, pp. 106–120, 2011. Springer-Verlag Berlin Heidelberg 2011.
18. Lauesen, S.: Soren Lauesen's website: www.itu.dk/people/slauesen/ Contains examples of requirements and user interfaces developed with tasks.
19. Lilly S.: Use Case Pitfalls: Top 10 Problems from Real Projects Using Use Cases, IEEE Computer Society, Washington, DC,US (1999)
20. Lim, K. Y.: Structured task analysis: an instantiation of the MUSE method for usability engineering. Interacting with Computers, vol 8, no. 1, pp. 31-50, 1996.
21. Maiden, N. A., Ncube, C.: Acquiring COTS software selection requirements. IEEE Software, March/April, pp. 46-56 (1998)
22. Polson, P.G., Lewis, C. H.: Theory-based design for easily learned user interfaces. Human-Computer Interaction, vol 5, pp. 191-220, 1990.
23. Preece, J., Rogers, Y. & Sharp, H.: Interaction Design – Beyond Human-Computer Interaction, John Wiley & Sons, New York, 2002.
24. Robertson, S. & Robertson, J.: Mastering the Requirements Process. Addison-Wesley, 1999.
25. Rogers, E. M.: Diffusion of Innovations. New York: Free Press (1962 and 1983).
26. Rosenberg, D., Scott K.: Top Ten Use Case Mistakes, Software Development (2001): <http://www.drdoobs.com/184414701>
27. Sigurðardóttir, Hrönn Kold, project manager for the Electronic Health Record system at the Capital Hospital Association (H:S): Draft of PhD thesis (2010)
28. Wiegers, K. E.: Software requirements. Microsoft Press, 2003.

29. Wirfs-Brock, R.: Designing Scenarios: Making the Case for a Use Case Framework, Smalltalk Report, Nov/Dec (1993). Also:
<http://www.wirfs-brock.com/PDFs/Designing%20Scenarios.pdf>
30. Yourdon, E.: Modern Structured Analysis. Prentice Hall, New Jersey, 1989.