# Tasks & Support
## Task Descriptions as Functional Requirements

Soren Lauesen

*IT University*
*Glentevej 67, DK-2400 Copenhagen NV*
*slauesen@itu.dk*

**Abstract**
This paper shows how task descriptions (a kind of use cases) can serve as verifiable requirements that the user easily can understand. Properly used, they avoid a premature division of work between user and computer, which is particularly important when buying COTS-based products. An extended version of task descriptions (Tasks & Support) furthermore allow a structured comparison of present user problems against possible solutions. We have used the techniques in many kinds of real-life projects and compare our experiences against traditional forms of requirements

**Keywords:** tasks, use cases, COTS, validation, method testing in real-life.

## 1. Introduction

There are many ways to write functional requirements, but most of them somehow describe the possible input to the system, the output from the system, and the relation between the two. In the following we will use a hotel administration system as an example. One of the requirements could be this:

**Feature requirement**

R1: The system shall record check-in of guests, and automatically allocate free rooms to them.

This traditional requirement style says what the computer system shall do. In many cases, this approach is not suitable because we have prematurely selected a division of work between the computer system and the user. Later in the project, we might for instance realise that automatic allocation isn't good for various reasons, and then we have to choose between developing an inferior system or changing the requirements. We might also be interested in not developing the system ourselves, but buying a COTS system through a tender pro??cess. If the best available system doesn't allocate rooms automatically, our requirement would be a barrier.

Are there any alternatives? Could we specify verifiable requirements without specifying what the computer system shall do? Yes, we can - at least to a large extent. The trick is to specify what the computer and the user shall do together, without caring about how the work is divided between the two actors. We will call this specification a *task description*. One of the requirements could then be this:

**Task-based requirement**

R2: The system shall support the check-in task described in . . .

Whether this is a good requirement depends of course on how we describe the tasks. Below we will describe them in ways inspired by Cockburn's use cases [2, 3]. For reasons to be explained in section 4, we prefer to talk about tasks rather than use cases.

Many authors have described how use cases, tasks, scenarios, etc. are used in development [e.g. 1, 3, 5, 11]. Here we will focus on how they can be used for requirements.

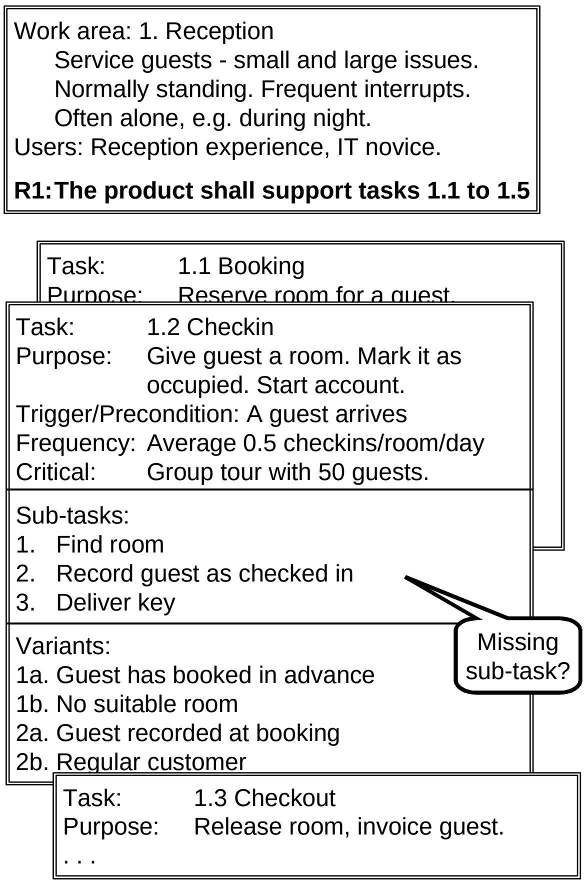
Given a developed system, can we verify whether it meets requirement R2? Yes, we can see whether it actually supports the described task, but we have to exercise a quality judgement to see how good the support is.

Below we will show detailed requirements that use task descriptions. We use a hotel system as an example because most people can imagine such a system. However, the example scales up to large systems. Over the last couple of years, I and my colleagues have successfully used task descriptions as requirements in real-life projects, including large COTS-based systems, small in-house systems, and product development for international markets.

## 2. Task descriptions

Figure 1 shows our version of task descriptions, illustrated by a hotel system. In addition to the tasks themselves, the description also has background information for the entire work area.

Figure 1. Task descriptions


Work area: 1. Reception
Service guests - small and large issues.
Normally standing. Frequent interrupts.
Often alone, e.g. during night.
Users: Reception experience, IT novice.

**R1: The product shall support tasks 1.1 to 1.5**

| | |
|---|---|
| Task: | 1.1 Booking |
| Purpose: | Reserve room for a guest. |

| | |
|---|---|
| Task: | 1.2 Checkin |
| Purpose: | Give guest a room. Mark it as occupied. Start account. |
| Trigger/Precondition: | A guest arrives |
| Frequency: | Average 0.5 checkins/room/day |
| Critical: | Group tour with 50 guests. |

Sub-tasks:
1. Find room
2. Record guest as checked in
3. Deliver key

Variants:
1a. Guest has booked in advance
1b. No suitable room
2a. Guest recorded at booking
2b. Regular customer

| | |
|---|---|
| Task: | 1.3 Checkout |
| Purpose: | Release room, invoice guest. |

. . .

### Work area

The example shows only the work area *reception*. A realistic hotel system would also support work areas such as staff scheduling, room maintenance, and accounting.

The work area description explains the overall purpose of the work, the work environment, the user profile, etc. You might wonder whether this information is requirements. As it appears in the example, it is not. It is background information that helps the developer understand the application domain. No matter how complete we try to make the specification, most real-life design decisions are based on developer intuition and creativity. The background information sharpens the developer's intuition.

In the example, the background information tells us that the system should support several concurrent tasks because there are frequent interrupts; a mouse might not be ideal when standing at a reception desk; allowing computer games or Web access during night shifts might be an advantage to keep the receptionist awake, etc. Depending on the kind of project, you might replace some of this with explicit requirements.

The work area description is the common background information for all the tasks in that work area. Use case specialists rarely use separate work area descriptions, but give some description of the user (the actor) for each task or use case. This duplicates information because the same users perform many tasks. As a result, the background descriptions tend to be short. Collecting them in a work area description encourages a more thorough description.

### Individual task descriptions

Below the work area description, we find descriptions of the individual tasks in a form similar to Cockburn's use cases. Each task has a specific goal or purpose. The user carries out the task and either achieves the goal or cancels the whole activity.

In the example, we recognise the booking, check-in, and check-out tasks. Let us look at check-in in detail.

**Purpose.** The purpose of check-in is to give the guest a room, mark it as occupied, and start the accounting for the stay. This translates well into state changes in the database. If the user cancels the task, there should be no traces in the database.

**Trigger/Precondition.** The template has space for a trigger or a precondition. A trigger says when the task starts, e.g. the event that initiates it. For check-in, the trigger is the arrival of a guest – he reports at the reception desk.

A precondition is something that must be fulfilled before the user can carry out the task. In the check-in case we have specified a trigger, but not a precondition. There is rarely a need for both, and in general we find that preconditions are rarely needed for tasks while they are important for use cases.

**Frequency and critical**. The fields for frequency and critical are very important in practice. The requirement on Figure 1 is to support 0.5 check-ins per room per day, and support critical activities with 50 guests arriving. What can that be used for in development?

Imagine 50 guests arriving by bus and being checked in individually. Imagine that each guest reports at the reception desk, the receptionist finds the guest, prints out a sheet for the guest to sign, and then completes the check-in of that guest. This could easily take over a minute per guest. The last guest will be extremely annoyed at having to wait one hour! Maybe we should provide some way of printing out a sheet for each guest in advance with his room number on it?

What about 0.5 check-ins per room per day? How many rooms are there? Well, a large hotel has 500 rooms, meaning that there are approximately 250 check-ins per day, with most guests probably arriving in peak hours. We definitely need a multi-user system – so that the system can deal with concurrent check-ins and ensure that no two customers end up being assigned to the same room. We can derive several design constraints from these two lines.

**Sub-tasks**. The central part of the task description is the list of sub-tasks. The receptionist must find a suitable room for the guest, record guest data, and record that the guest is checked in and the room occupied. Finally he must give the guest the room key.

These sub-tasks specify what the user and the computer must do together. Who does what depends on the design of the product or on the chosen COTS system. What about the sub-task *Deliver key?* Should that be computer-supported too? Maybe. Some hotel systems provide electronic keys, unique for each guest, but that is expensive. Obviously the solution has to be decided later in the project, depending on the costs and benefits involved.

One of the advantages of task descriptions is that the customer readily understands them. If we try to validate the check-in task with an experienced receptionist, he will immediately notice that something important is missing: ‘In our hotel, we don't check guests in until we know they can pay. Usually we check their credit card, and sometimes we ask for a cash deposit. Where is that in your task description?’

"Oops" said the analyst and added this line between sub-task 1 and 2:

2. Check credit card or get deposit

**Variants**. Finally, there is a list of variants for the sub-tasks.

Sub-task 1 (find room) has two variants: (1a) The guest may have booked in advance, so a room is already assigned to him; (1b) There is no suitable room (suggests some communication between receptionist and guest about what is available, prices etc.).

Sub-task 2 (record guest) also has variants: (2a) The guest may have booked in advance and is thus recorded already. (2b) He is a regular customer with a record in the database.

Variants are a blessing for analysts. You don't have to describe rules or specify logic for the many special cases; simply list the variants to be dealt with.

**Task sequence**. Although the sub-tasks are enumerated for reference purposes, no sequence is prescribed. In practice users often vary the sequence. It is a good idea to show a typical sequence, but it doesn't mean that it is the only one.

**Development and verification**

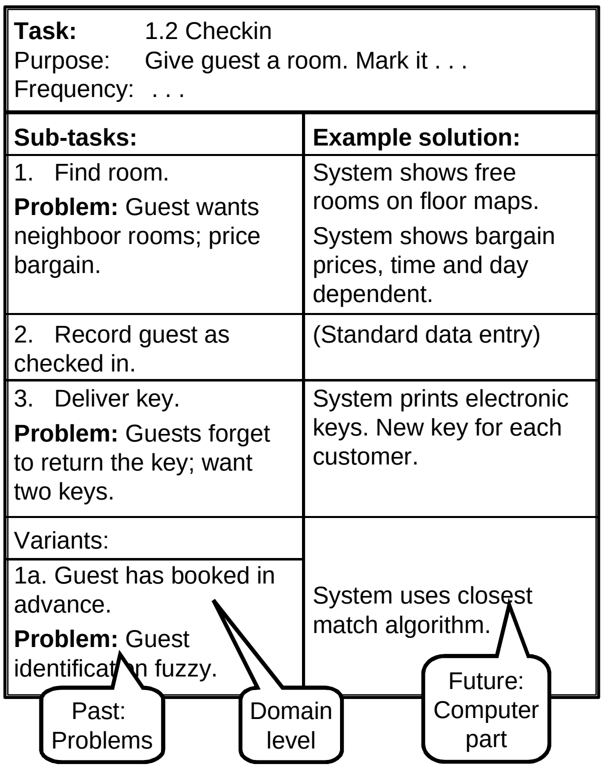
How can a task description of this kind be used during development and at delivery time? Although customers as well as developers easily understand task descriptions, there is a larger gap to design and development than with traditional feature requirements. The developers have to be more innovative to find good ways of supporting the tasks, and the responsibility to do so rests with them.

However, once a design is suggested, it is easy to check that it supports the tasks. The developers can simply simulate that they carry out the tasks and all their variants. Verifying the requirements at delivery time is a matter of having the user carry out the tasks and the variants.

Actually, the design problem is not that hard. Lauesen & Harning [8] explain a systematic way to handle it. In summary, the developers first design screens that give the user the necessary data for each task or sub-task, trying at the same time to keep the number of screens low. Second, the developers add the necessary functions for carrying out the tasks, resulting in a mockup or prototype of the interface. Finally, they usability test the user interface, modify it as needed, and implement it. The usability test essentially serves as an early verification that the tasks are supported in an efficient way.

Developers that follow this approach have realised that the traditional object-oriented analysis doesn't help in this design process (except for the datamodel aspect of the classes). On the contrary, it slows down the pro??cess. The datamodel and the task descriptions are a sufficient basis for the design. Programming may - de-

Figure 2. Tasks & Support



| | |
|---|---|
| **Task:** 1.2 Checkin<br>Purpose: Give guest a room. Mark it . . .<br>Frequency: . . . | |
| **Sub-tasks:** | **Example solution:** |
| 1. Find room.<br>**Problem:** Guest wants neighboor rooms; price bargain. | System shows free rooms on floor maps.<br>System shows bargain prices, time and day dependent. |
| 2. Record guest as checked in. | (Standard data entry) |
| 3. Deliver key.<br>**Problem:** Guests forget to return the key; want two keys. | System prints electronic keys. New key for each customer. |
| Variants: | |
| 1a. Guest has booked in advance.<br>**Problem:** Guest identification fuzzy. | System uses closest match algorithm. |

pending on the programming language used - be object-oriented or not.

## 3. Tasks & Support

The ideal task descriptions are independent of the division of work between computer and user, but are they also independent of whether we talk about the past or the future? Is the difference between how we carried out the tasks before and how we want to carry them out in the future merely a matter of a new way of dividing the work? In principle yes, but in practice it turns out to be advantageous to identify problems in the old way of doing things and outline new ways of doing them in the future.

Tasks & Support is a systematic way of dealing with this. Figure 2 shows a Task & Support description of the check-in task. It consists of several parts:

**Domain-level activity**. The left-hand column describes the domain-level activity – what human and computer do together. In the example the description is very short; simply the name of the sub-task. In real specifications, a few lines are sometimes needed.

We suggest that imperative language should be used here, e.g. *Find room*, to hide whether a human or a computer carry out the sub-task.

**Problem**. The problem part is the only part of the description that mentions something about what happens in the old system. You only specify problems if there are any. As an example, sub-task 2, *Record guest*, doesn't have any significant problems, so a description of the domain-level activity suffices.

Note that the problem part gives us an opportunity to specify things we cannot specify in more traditional requirements. For instance, the problems in sub-task 1 show that automatic room allocation is a bad idea.

**Solution**. In the right-hand column we outline how the new system could support the activities and how it could solve the problems. This part shows something about the future and what the product should do. In practice this is an area for discussion between customer and supplier, who should try to arrive at an agreement based on the benefit to the customer and the cost of providing the solution.

Figure 2 shows *example solutions* as indicated by the right-hand column heading. In a later version, the supplier may change this column to reflect new ideas or proposals, and the heading should change to 'Proposal'. Finally, the column is changed to what the two parties eventually agree to provide, and the heading should change accordingly to ‘Agreement’.

To emphasize the computer aspect, we suggest that statements with an explicit subject should be used, e.g. *System shows free rooms* or *Product shows free rooms*. Traditional wording such as *The system shall show free rooms* may be used if you like, but only if the parties decide that the right-hand column is the requirement, thus giving up the explicit requirement to support the user tasks and solve the user problems.

Figure 2 shows various non-trivial solutions to the problems. For instance, some hotels may be willing to negotiate a discount if the customer arrives in the afternoon and the hotel has many vacancies. The system could guide the receptionist in these matters. Perhaps one supplier has realised that the weather has an influence on such negotiating, since customers would be more reluctant to go to other local hotels in rainy weather, so he offers a feature for entering weather conditions, thus exceeding the customer's expectations. The supplier specifies this proposal in the solution column.

In some cases the solution is trivial. Sub-task 2, for instance, calls for ordinary data entry only; nothing needs to be specified. Many sub-tasks in real systems are trivial data entry tasks. In these cases there is not much difference between the domain-level activity, the user activity, and the computer activity. The feature requirements are trivial.

See Lauesen [9] for further explanation of the technique, how to compare proposals, etc.

## 4. Use cases versus tasks

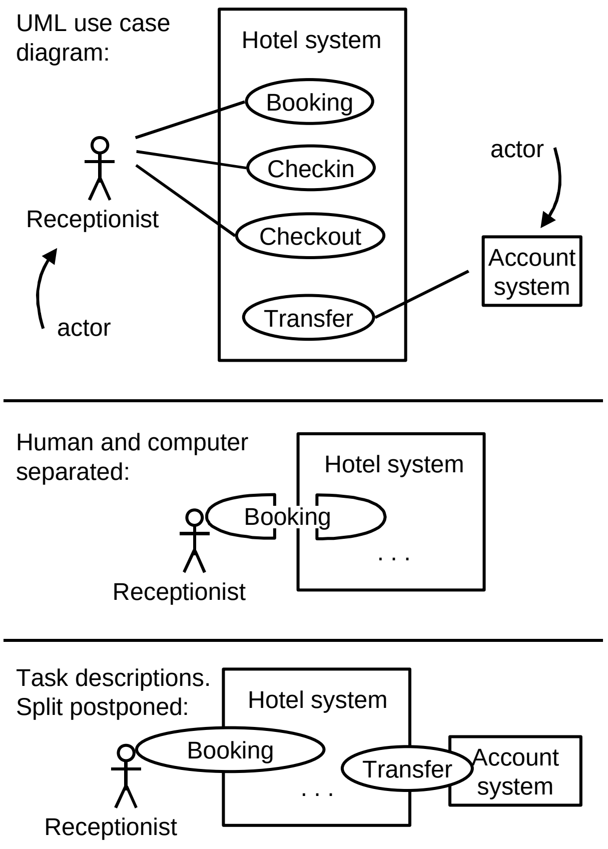


Figure 3. Use cases vs. tasks

A task is what human and computer do together. In contrast, a use case is primarily the computer's part of a task, including its interaction with the user. Use cases were introduced by Jacobson et al. [6] as a literal translation from Swedish, and the term is now used extensively in connection with object-oriented software development (Booch et al. [1]; Stevens and Pooley [10]). However, the term *use case* has been used in so many ways that it is hard to know what people are really talking about when they use it (Cockburn [2]; Constantine and Lockwood [4]).

We will first look at the UML version of use cases. UML definitions of use cases have changed over time, but here is a recent definition by Booch et al. [1]:

*A use case is a description of a set of sequences of actions, including variants, that a system performs to yield an observable result to an actor*.

Note that the definition only talks about the actions performed by the system (the computer), not the actions performed by the user.

The first diagram on Figure 3 is a UML diagram of four use cases. The box represents the computer system and the diagram shows that the receptionist can carry out (be the *actor*) of the use cases *booking*, *check-in*, and *check-out*. These use cases are handled by the computer system, as illustrated by the bubbles inside the box. Each use case bubble might involve several system functions, for instance listing free rooms, selecting rooms, and recording guest information.

Note that the accounting system is an actor too. We assume that accounting is handled as data transfer to a separate accounting system. The bubble shows the hotel system's part of the transfer.

The graphical representation suggests that the use case is something done by the computer, not something done by user and computer together. The figure reflects the current thinking that use cases are computer-oriented.

However, there are other types of use cases than the one defined in UML. In the second diagram in Figure 3, we have illustrated a kind of use case where we can see user actions as well as computer actions (one example is Constantine & Lockwood's *essential use cases* [4, 5]). The diagram shows that the entire booking task consists of two parts, one carried out by the user and one carried out by the product.

The last diagram in Figure 3 illustrates the task concept. The bubble represents the entire task. It floats over the product boundary, illustrating that the task is carried out by human and computer together, but the division of labour is not yet determined. The transfer task also has a hotel system part and an accounting system part, with a division not yet determined.

Figure 4. High-level tasks

| | |
|---|---|
| **Task:** **1. A stay at the hotel**<br>Actor: The guest<br>Purpose: . . . | |
| Sub-tasks: | Example solution: |
| 1. Select a hotel.<br>**Problem:** We aren't visible enough. | ? |
| 2. Booking.<br>**Problem:** Language and time zones. Guest wants two neighbor rooms | Web-booking.<br>Choose rooms on web at a fee. |
| 3. Check in.<br>**Problem:** Guests want two keys | Electronic keys. |
| 4. Receive service | |
| 5. Check out<br>**Problem:** Long queue in the morning | Use electronic key for self- checkout. |
| 6. Reimburse expenses<br>**Problem:** Private services on the bill | Split into two invoices, e.g. through TV. |

## 5. High-level tasks

Above we have assumed that the same users will carry out the tasks in both the old and the new solution. What should we do if this assumption isn't valid, or if we plan an entirely new system without present users? A good approach is to look at the situation as seen from the client's viewpoint. In the hotel example, the receptionist is the user and the guest is the client.

If we look at the hotel from the guest's point of view, staying at the hotel is a kind of task. It is not a traditional human-computer task since the guest may not interact directly with the computer, but it is an interesting task anyway, because the ultimate success of the system depends on how well it serves the guests.

Figure 4 shows the sub-tasks of a hotel stay as seen by the guest. Now we see the previous tasks, Book, Check-in, etc., as sub-tasks of this high-level task.

We also see two new tasks: select a hotel and reimburse expenses. Are they of interest when defining the requirements? They may very well be. For instance, a business guest needs an invoice to claim reimbursement, but some of his expenses will not be reimbursable and it simplifies matters to the guest if they do not appear on the main invoice. (In fact, some expenses might be outright embarrassing to have on the main invoice!)

Our preoccupation with the receptionist has so far prevented us from seeing the customer's needs, so a high-level task description will help us to see the key business needs. We can use the high-level task as an analysis tool to reveal additional requirements. In this case we identified a need for separating reimbursable expenses from other expenses. We could add it as a feature requirement or we could state it as a problem in the Task & Support description for check-out.

Business process re-engineering uses radical restructuring of a company to better serve the clients and reduce costs. The present user tasks are not taken for granted. Some of them may disappear, others are redefined, and new ones may come up. High-level task descriptions can help in that process. For instance, we might ask whether we could support the hotel-stay task any better.

In the example, we first identified the customer's sub-tasks and problems. In a later brainstorming session, we came up with possible solutions to some of the problems.

The general trend in the solutions is to allow the customer to do more for himself. We could help him to book through the Internet, and why not allow him to select a room too? We could also allow him to order services electronically during his stay. He could check out by inserting his electronic room key into a slot at the reception desk, thus bypassing the morning queue of other guests checking out.

## 6. A hospital case

The Task & Support idea was developed by the author and Marianne Mathiassen in close co-operation with a large customer (West Zealand hospital) and three COTS suppliers [7].

The hospital had experienced severe problems when acquiring systems through tender processes, and we studied what had happened and how it related to the requirements. As a result of this study we came up with the Task & Support idea, and we wanted to test the idea on a realistic scale in the same organisation.

We took an existing hospital system recently contracted with a supplier but not yet delivered, and developed Tasks & Support for the most difficult application area: roster planning. This was also the most business-critical area because many savings were expected from improved roster planning. Modelling this area required eight task descriptions.

Figure 5 shows an abbreviated version of the most critical and complex task: allocating duties to staff. The task is actually carried out over a period of several days where the user tries to allocate staff and get feedback from others about the allocation. Some of the sub-tasks are carried out several times during the total planning task.

Figure 5. Hospital roster planning

| | | |
|---|---|---|
| **Task** | **1.2 Make roster** | |
| Goal | Staff all duties. Ensure regulations . . . Ensure low cost | |
| Frequency | Once every two weeks. In some departments . . . | |
| Critical | Vacation periods . . . | |
| **Sub-tasks:** | | **Example of solution:** |
| **1** | **Initialize new roster period** | System generates roster for new period based on . . . |
| **2** | **Record staff leave**<br>Two kinds of leave: . . .<br>**Present problems:**<br>Leave requests kept on manual notes, often months into the future. | System can record leave one year into the future. System warns if leave is against regulations.<br><br>It must be easy to record a long period of leave (several months). |
| **3** | **Allocate staff for unstaffed duties.**<br>Ensure level of competence, regulations, leave days, and low cost.<br>**Present problems:**<br>Difficult to ensure this manually. Costs are higher than necessary and errors occur. | System shows unstaffed duties and suggestions for staffing. User selects the actual staff. System warns if duties are unstaffed, leave or regulations violated, or cost unnecessary. Warnings must be immediate to support the puzzle.<br><br>System supports extensive undo and several temporary versions. |
| **4** | **Send roster for review** | A print of the roster is sufficient. |
| **5** | **Modify roster** | Steps above suffice |
| **6** | **Authorize roster** | . . . |
| **Variants:** | | **Example of solution:** |
| **3a** | **Staff not yet recorded in the staff file** | User enters preliminary data for new staff. |
| **3b** | **No staff available**<br>**Present problem:** No information about staff in other departments | System suggests staff from other departments based on their authorized rosters. |

Note sub-task 3, Allocate staff for unstaffed duties. It is the most critical part because most of the economic advantage must be obtained there. Although the monetary benefits are not shown there, you can clearly see that this is a very important sub-task.

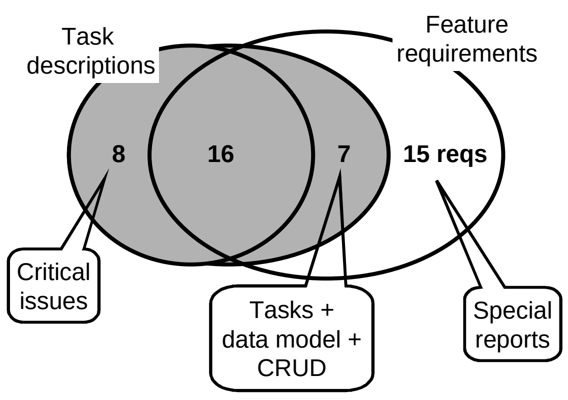
Another interesting thing is variant 3b, No staff available. The user (planner) works for a single hospital department and doesn't have information on staff in other departments. Because qualified staff are becoming scarcer, the users dreamt of getting on-line information about available staff in other departments. They reasoned that the system ought to know the roster for other departments too, so it should be able to list available staff whom they might call on for additional help.

When we later checked the approach with three suppliers, one of them laughed at this ‘primitive requirement’. His company provided computer services for all the local hospitals, and they could easily provide access to available staff there too. The requirements format enabled him to tell the customer that he could exceed his dreams.

### What is covered by the technique?

To what extent can Tasks & Support replace traditional functional requirements? We checked that in the hospital where we developed the technique. When we had developed the Task & Support requirements in co-

Figure 6. Match with old feature spec



operation with the expert users, we compared the 38 original, feature-based requirements with those expressed through the task descriptions. Figure 6 shows a summary of the results.

Sixteen original requirements were covered by the task descriptions.

Seven original requirements were not covered by the task descriptions, but we estimated that they would have been if we had made a data model and cross-checked it against the tasks. For instance, we had overlooked that the system had budgets for each department, and some tasks dealing with budgeting were needed to provide the budget data.

Fifteen original requirements were not covered, and we had difficulty seeing how this could have been done. They all specified some special report to be produced, and the old system had these features. However, nobody we talked to could explain what these reports were used for, so it was impossible for us to identify any tasks needing these requirements. We believe that some of them were actually used somewhere, while others were just relics of the old system.

Eight requirements were new. The task descriptions clearly showed a need for these eight things, but they were not mentioned in the old requirements. All were critical in some task. Half happened to be provided by the supplier anyway, but the rest were not. This led to great consternation in the IT department as these deficiencies were realised, particularly because some of the business goals could thus not be met.

The conclusion is that Tasks & Support can reveal critical requirements that can otherwise be easily overlooked. However, some functional requirements are hard to catch in this way, because they don't clearly relate to tasks. Unfortunately we cannot point to any single technique that would reveal these missing requirements.

**Cost of the technique**

The Task & Support technique doesn't require a lot of time, training or tool support; but it needs close guidance by an expert. We will illustrate the typical pattern with what happened in another project.

The hospital decided to use Tasks & Support in a new COTS acquisition, possibly with tailor-made extensions. The application was about patient administration across all departments. The value of the contract was expected to be approximately US$4 million plus around $2 million per year for operating the system. Here is a brief account of the work.

The author, working as a consultant, trained two expert users and one IT specialist for two days. They had been involved in traditional feature-oriented requirements before, but had never seen task or use case techniques.

As part of the two days of training, they outlined a single high-level task that covered most of the system as seen from the patient's point of view. This outline used nine ordinary task descriptions to be specified later.

Next, the two expert users worked alone specifying some of the ordinary tasks. After some initial mistakes, which the consultant helped them correct, they completed the entire spec in ten more days, including reviews in the departments. Then they sent it for a blitz review by the consultant. He was truly impressed. They had not only made excellent task descriptions, but they had also found a creative way to use the same template for non-task issues, such as maintenance, daily operation, and usability.

The consultant had only minor comments, and the spec was sent out for tender. Total man days: around 25. Total consultancy days: 3. Seasoned developers seem to learn the technique even faster. We have seen them master it in a single day.

**Improved requirements writing**

The hospital team later reported that similar projects used to take 25 weeks with feature-based requirements, rather than the three weeks with the new approach. Furthermore, the new approach ensured that they got what they needed. Figure 7 summarises the differences.

Previously, the IT department had asked each user department (wards, labs, personnel department, etc.) to write down their requirements, and the IT department then edited the whole thing and sent it for comments and approval in the departments. This caused a lot of debate on whether the spec was complete and whether this or that was needed.

Amazingly, although the IT department had edited the spec, they often didn't understand the requirements, but assumed that the user departments knew what they had asked for and that the supplier would also know. Our later talks with the suppliers revealed that they too

Figure 7. Early experiences

| Traditional | Tasks & Support |
|---|---|
| **Write requirements**<br>Everybody asked.<br>All dream up reqmts.<br>Combined into one spec.<br>Few understand it. | **Write requirements**<br>Expert users describe tasks.<br>Everybody can correct tasks and add *wishes*. |
| Time: 25 weeks | Time: 3 weeks (first time) |
| **Assess proposals**<br>Everybody has a say | **Assess proposals**<br>Carry out the tasks - give scores. |
| Political choice | Selected stakeholders asked. No doubt. |
| Time: 10 man months | Time: 1 man month |

weren't sure what the hospital asked for, but assumed it could be resolved during the project.

With the Task & Support approach, a small group of expert users, assisted by the IT department, wrote a set of task descriptions, sometimes with suggested solutions. The expert user's deep task understanding was a key factor in the approach. Then they sent it off for comments and approval in the user departments as usual. The departments now commented primarily on the completeness of the task descriptions, which are facts, rather than on the required features, which tend to be a matter of opinion. When a department suggested some solutions, they were simply included as possible solutions in the right-hand column, i.e. as an example rather than a requirement.

It should be mentioned that in this example, the hospital had a fairly good idea what kind of system they wanted and what kind of business goals to go for. There was also a reasonable commitment by all departments involved. This had been the case both when the old feature-based method had been used, and when Tasks & Support had been used.

In other organisations, goals and commitment may be serious issues. Resolving them may take a long time, making it difficult to see the full effect of the Task & Support approach.

#### Comparing proposals

Comparing the suppliers' proposals also went much more smoothly than usual. The team spent 20 man days comparing the two best proposals in detail. Essentially they made a kind of acceptance test of the existing versions of the products, working through all the task descriptions and variants to see and describe how well the systems and the promised extensions would support them. Their comparison convinced stakeholders without further discussion. The traditional approach required ten times as much work because many stakeholders had to review and comment on the proposals.

## 7. Conclusion

### Advantages of task-based requirements

The comparison below is based on experiences from many types of real-life projects, e.g. product development, tailor-made systems, and COTS based acquisition.

**Validation, verification, etc.** The customer can easily validate task descriptions and ensure that he gets what he needs. Developers can better understand the requirements and check that their design is adequate. Finally, the parties can easily verify the requirements during and at the end of development.

**Product development.** Tasks & support help the developers identify the important features to be developed. High-level tasks have repeatedly given rise to innovative products with excellent market acceptance.

**Tenders**. We have much experience with task & support requirements in tender processes, where the customer announces a request for proposal and several suppliers reply. Whether we are dealing with COTS-based products or tailor-made products, suppliers as well as customers report these advantages:

1. It is much easier than usual to understand what the customer really needs and what kind of solution he has in mind.
2. It is possible to trace between requirements and business goals [7, 9].
3. The supplier can specify the advantages of his solution by relating it to the user tasks, and he can also show where his solution exceeds the customer's expectations.
4. The supplier can demonstrate to the customer how the tasks will be supported, and how the critical issues will be handled.
5. All suppliers get equal opportunities since no solution is prescribed.
6. It is possible to adjust ambitions in the solution according to needs and costs.

### Disadvantages of task-based requirements

**No data specified, non-task activities**. Little is shown about the data required for the tasks. Also, some activities are hard to describe as tasks.

**More work for the COTS supplier?** Some COTS suppliers are concerned that the task-based approach takes longer than traditional approaches. Previously, they could just cut and paste from other proposals. With task-based requirements they have to understand the user's tasks, they complain. This is true, but a clever customer will insist on task descriptions for just that reason.

**More work for the developer?** When the system is to be developed from scratch, there is a longer jump to the solution than with traditional feature requirements. On the other hand, the task descriptions better ensure that the solution actually meets the real demands.

**More work for the customer?** Some suppliers suggest that it is more laborious for the customer as well. In our experience, this is not true. The specification work is actually reduced drastically compared to traditional specifications. Of course, compared to the approach where the customer doesn't specify anything but leaves it to the supplier to set up a specification, it is more laborious.

**Unusual reply format**. In tender processes, we have found that many suppliers hesitate to modify the right-hand column to show their solution. They prefer to specify their solution in attachments. This, however, makes it more difficult for the customer to evaluate the proposals. Skilled suppliers modify the text in revision mode, thus clearly showing what they have changed. They sometimes attach product descriptions, for instance screen pictures, and then refer to them from the task description.

When supplier and customer jointly develop Tasks & Support, there is no such problem with modifying the description.

## Acknowledgements

Large parts of this paper are copied from the author's book "Software Requirements" [9], with permission from the publisher, Pearson-Addison Wesley.

### References

1. Booch, G., Rumbaugh, J. & Jacobson, I. (1999) The Unified Modelling Language. User Guide. Addison-Wesley.
2. Cockburn, A. (1997) Structuring use cases with goals. Journal of Object-Oriented Programming, Sep-Oct, 35–40 & Nov-Dec, 56–62. Also in: http://members.-aol.com/acockburn/papers/usecases.htm.
3. Cockburn, A. (2000) Writing Effective Use Cases. Addison-Wesley.
4. Constantine, L. & Lockwood, L.A.D. (2001) Structure and style in use cases for user interface design. In Object Modelling and User Interface Design (ed. M.V. Harmelen), Addison-Wesley.
5. Constantine, L. & Lockwood, L.A.D. (1999) Software for Use: A practical guide to the models and methods of usage-centered design. Addison-Wesley.
6. Jacobson, I., Christerson, M., Jonsson, P. & Övergaard, G. (1994) Object-oriented Software Engineering – a use case driven approach. Addison-Wesley.
7. Lauesen, S. & Mathiassen, M. (1999) Use cases in a COTS tender. In Proceedings of the Fifth International Workshop on Requirements Engineering (eds A.L. Opdahl, K. Pohl and E. Dubois), REFSQ'99, Presses Universitaires de Namur, 1999, 115–129.
8. Lauesen, S. & Harning, M. B. (2001) Virtual Windows - Linking user tasks, data models, and interface design. IEEE Software, July/August, 67-75.
9. Lauesen, S. (2001): Software Requirements - Styles and Techniques. Pearson-Addison Wesley (in print).
10. Stevens, P. & Pooley, R. (2000) Using UML, Software engineering with objects and components. Pearson Education, London.
11. Weidenhaupt, K., Pohl, K., Jarke, M. & Haumer, P. (1998) Scenarios in system development: Current practice. IEEE Software, March/April, 34–45.