Task Descriptions as Functional Requirements

Soren Lauesen, *IT-University of Copenhagen*

Requirements form a software system's foundation. Functional requirements indicate what the system shall do, data requirements indicate what it shall store, and quality requirements indicate how quickly or how easily it shall perform. This article focuses on functional requirements, which usually describe a system's input, output, and the relationship between the two. Traditional functional requirements specify the system's role but ignore the system's context. To solve that problem,

Marianne Mathiassen and I developed the *Tasks & Support* method, which uses annotated *task descriptions*. They specify what the computer and user shall accomplish together without indicating which actor performs which parts of the tasks. This method produces higher-quality requirements that are faster to produce and easy to verify and validate.

Traditional requirements versus task descriptions

Let's look at two specific requirements for a hypothetical hotel-administration system. The system could have this feature requirement:

R1: The system shall record guest check-in and automatically allocate free rooms.

This traditional requirement style says what the computer system shall do. This approach is often unsuitable because it prematurely divides work between the computer and the user. If we used this style, we might discover late in the project that automatic

allocation won't work. Then we'd have to choose whether to develop a subpar system or to change its requirements. Premature work allocation might also handicap us if we purchased a COTS (commercial off-the-shelf) system instead of developing our own. The best available system might not allocate rooms automatically, so our requirement would be a barrier.

Now let's examine a task description. A single task-based requirement could cover most of the functional requirements:

R2: The system shall support all the tasks described in …

Whether this is a good requirement depends on how we describe the tasks. The Tasks & Support method describes them in ways inspired by Alistair Cockburn's *use cases*. Use cases tend to describe what the system does and how it interacts with the user. However, we want to delay splitting the work between the system and the user, because this is a design decision to be made later. So, we use

Use Cases versus Task-Based Descriptions

Use cases were introduced by Ivar Jacobson and his colleagues,[1] and the term is now used extensively for object-oriented software development. However, people have used use case in so many ways that it is hard to know what they are really talking about.[2–5] Figure A contrasts two use case versions against task descriptions, using a hypothetical hotel-administration system.

Figure A1 shows the Unified Modeling Language version of use cases. UML use case definitions have changed over time, but recently Grady Booch, James Rumbaugh, and Ivar Jacobson offered the following:

> *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.*[6]

The definition only addresses the actions that the system (computer)—not the user—performs. The UML diagram reflects this, showing use cases as bubbles inside the system.

Figure A2 illustrates a use case that distinguishes between user and computer actions. (Larry Constantine and Lucy Lockwood's *essential use cases* provide an example.[5,7]) In the figure, the booking task consists of two parts—one that the user performs and one that the system performs.

Figure A3 illustrates the task description concept. The bubble represents the entire task. It floats over the system boundary, illustrating that human and computer carry out the task together. This approach does not divide the labor—that is a design issue to be dealt with later.

References

1. I. Jacobson et al., Object-Oriented Software Engineering: A Use Case Driven Approach, Addison-Wesley, Boston, 1994.
2. A. Cockburn, "Structuring Use Cases with Goals" (Part 1), J. Object-Oriented Programming, Sept.–Oct. 1997, pp. 35–40; http://members.aol.com/acockburn/papers/usecases.htm.
3. A. Cockburn, "Structuring Use Cases with Goals," (Part 2), J. Object-Oriented Programming, Nov.–Dec. 1997, pp. 56–62; http://members.aol.com/acockburn/papers/usecases.htm.
4. A. Cockburn, Writing Effective Use Cases, Addison-Wesley, Boston, Mass., 2001.
5. L. Constantine and L. Lockwood, "Structure and Style in Use Cases for User Interface Design," Object Modeling and User Interface Design, M.V. Harmelen, ed., Addison-Wesley, Boston, Mass., 2001.
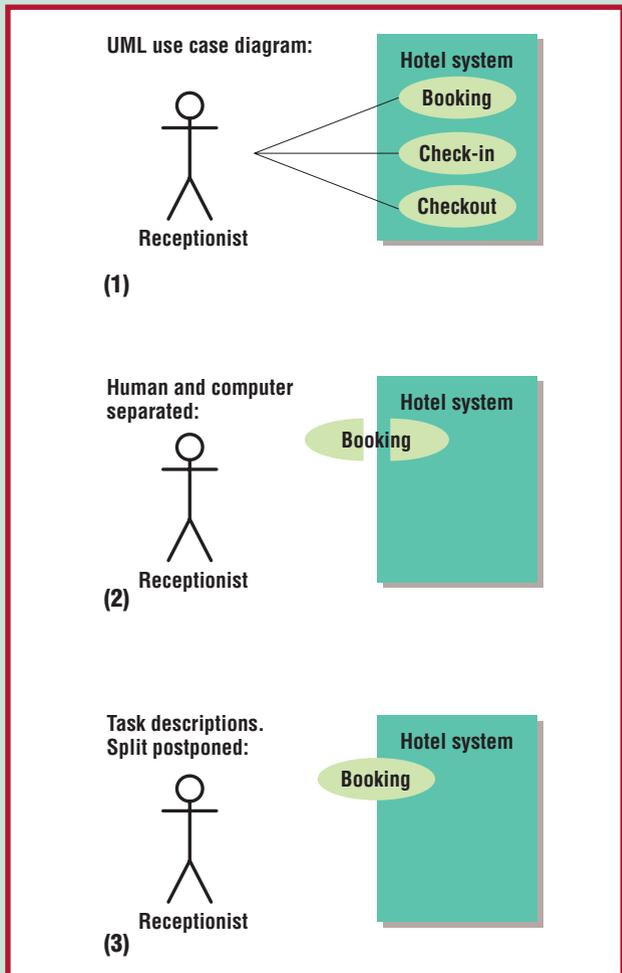
Figure A. Use cases versus tasks: (1) a Unified Modeling Language use case diagram, which deals only with the computer system's actions; (2) a use case that specifies separate human and computer actions; (3) task descriptions, which do not separate human and computer actions.

6. G. Booch, J. Rumbaugh, and I. Jacobson, The Unified Modeling Language User Guide, Addison-Wesley, Boston, Mass., 1999.
7. L. Constantine and L. Lockwood, Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design, Addison-Wesley, Boston, Mass., 1999.

the term task rather than *use case*. (For more on use cases and how they differ from task descriptions, see the related sidebar)

Can we verify that the system meets requirement R2? Yes, but we will have to judge how good the support is.
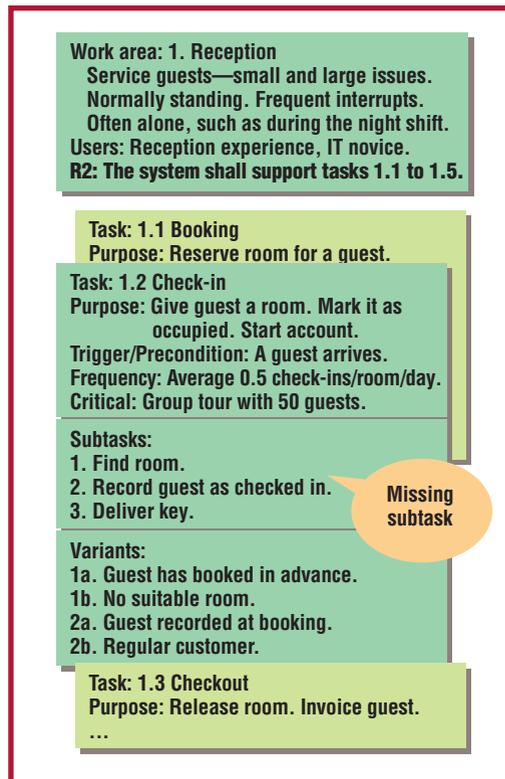
Task description details

Figure 1 shows task descriptions for the hotel's reception work area. A real hotel system would also support work areas such as staff scheduling, room maintenance, and accounting.

Work area description

The work area description states the formal requirement (R2: The system shall support tasks …). It also explains the work's overall purpose, the work environment, the user profile, and so on. As this information appears in the example, it is not requirements, but rather background information that helps the developer understand the application domain. No matter how complete we try to make the specification, developer intuition and creativity drive most real-life design decisions. The background information

Figure 1. Task descriptions for a hotel's reception work area.

> **Work area: 1. Reception**
>   Service guests—small and large issues.
>   Normally standing. Frequent interrupts.
>   Often alone, such as during the night shift.
> **Users: Reception experience, IT novice.**
> **R2: The system shall support tasks 1.1 to 1.5.**
>
> **Task: 1.1 Booking**
> **Purpose: Reserve room for a guest.**
>
> **Task: 1.2 Check-in**
> **Purpose: Give guest a room. Mark it as**
>   **occupied. Start account.**
> **Trigger/Precondition: A guest arrives.**
> **Frequency: Average 0.5 check-ins/room/day.**
> **Critical: Group tour with 50 guests.**
>
> **Subtasks:**
> 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.**
>   …

Missing subtask

sharpens the developer's intuition. Collecting the information in a work area description rather than writing something for each task encourages a more thorough description.

Individual task descriptions

Below the work area description are individual task descriptions. Figure 1 illustrates the booking, check-in, and checkout tasks. We'll look at check-in in detail.

Purpose. Check-in's purpose (also called *task goal* and *postcondition*) is to give the guest a room, mark it as occupied, and start accounting for the guest's stay. This process translates well into state changes in the database.

Trigger and precondition. The template has space for a trigger and a precondition. A trigger (or *business event*) indicates the event that initiates a task. For check-in, the trigger is a guest's arrival at the reception desk.

For check-in, we have specified a trigger but not a precondition. Analysts might consider specifying as a precondition that the guest must have a reservation or the hotel must have an available room. However, this is not actually a precondition because the receptionist begins the check-in task before he or she confirms reservations or room availability. The receptionist must cancel the check-in if the guest has not reserved a room and the hotel does not have one avail-

able. Good support for all these variants is crucial in the user–system dialogue.

Usually, checking conditions and business rules is part of the task, so specifying them as task preconditions makes little sense. For individual task steps, however, a precondition often makes sense.

Frequency and critical. The task description's `Frequency` and `Critical` fields are important requirements. In Figure 1, these requirements are to support

- On average 0.5 check-ins per room per day
- A group tour with 50 guests

How do these requirements affect system development? Imagine 50 guests arriving by bus and checking in individually. Each guest reports at the reception desk; the receptionist finds the guest in the system, prints out a sheet for the guest to sign, then completes the guest's check-in. This could take more than one minute per guest. The last guest will be extremely annoyed at having to wait almost an hour! The system could, for instance, provide a way for the receptionist to print in advance a sheet for each guest showing that guest's room assignment.

Subtasks. The `Subtasks` list is the task description's central part. The receptionist must find a suitable room for a guest, record guest data, and record that the guest is checked in and the room is occupied. Finally, the receptionist must give the guest the room key. Subtask 1 and Variant 1a address one of the preconditions I discussed.

These subtasks specify what the user and the computer must do together. Who does what depends on the system's design or on the COTS product the client chooses. Should the computer also support the subtask `Deliver key`? Maybe. Some hotel systems provide unique electronic keys for each guest, but that is expensive. The client must decide on a solution later in the project, depending on costs and benefits.

One advantage of task descriptions is that the customer readily understands them. If we validated the check-in task with an experienced receptionist, he or she would notice that something was 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 Subtasks 1 and 2:

```
2. Check credit card or get
   deposit.
```

Variants. Finally, we have the list of subtask variants. For instance, Subtask 1 (`Find room`) has two variants:

- 1a: The guest might have booked in advance, so the hotel has already assigned the guest a room.
- 1b: The hotel has no suitable room. (This suggests communication between the receptionist and guest about room availability, pricing, and so on.)

Variants relieve analysts from having to describe rules or specify logic for the many special cases; analysts simply list the variants. Later, the customer can easily verify that the system supports all variants.

Subtask sequence. Although we have enumerated the subtasks for reference purposes, no sequence is prescribed. In practice, users can vary the sequence. We show a typical sequence, but it is not the only one possible.

Tasks on different levels

Many practitioners have difficulty choosing appropriate task levels. They might, for instance, describe `Find room` as a separate task rather than a subtask. However, `Find room` is not a meaningful task because a task must bring closure—the user must feel he has achieved something when the task is complete. (The task should meet the coffee break test: The user deserves a cup of coffee after completing the task.) Describing each subtask as if it were a separate task is common among use case practitioners, but it leads to huge specifications that hide the system's essentials.

Also, from a requirements viewpoint, we want to strongly support each whole task. Providing strong support for each subtask but cumbersome transitions between subtasks is inadequate.

Development and verification

How can you use task descriptions during development and at delivery time? Al-

Figure 2. A Tasks & Support description of hotel check-in.

though customers and developers easily understand task descriptions, task descriptions require a larger leap from requirements to design and development than traditional feature requirements do. Developers must be more innovative to effectively support the tasks. (See the article on Virtual Windows for a systematic way to handle such innovation.[1]) However, once developers suggest a design, they can verify that it supports the tasks by simulating the tasks and all their variants. At delivery time, users will verify the requirements by performing the tasks and variants.

Tasks & Support descriptions

The ideal task descriptions are independent of the computer–user labor division and of time. In principle, the difference between how the actors have performed tasks and how they want to perform them in the future is merely a matter of dividing the work differently. In practice, however, we should identify problems in the old method and outline future solutions.

A *Tasks & Support description* is a systematic way to express problems and potential solutions. Figure 2 shows hotel check-in as an example. The description comprises several parts.

The left column describes the *domain-level activity*—what the human and computer do together. In this example, the description is simply the subtask name. Real specifications sometimes require a few descriptive lines. We

Figure 3. High-level tasks for innovation and business-process redesign.

Task:	Hotel stay	
Actor:	The guest	
Purpose:	…	
Subtasks:		**Example solution:**
1. Select a hotel. **Problem:** We aren't visible enough.		?
2. Book. **Problem:** Language and time zones. Guest wants two neighboring rooms.		Web booking. Choose rooms on Web for a fee.
3. Check in. **Problem:** Guest wants two keys.		Electronic keys.
4. Receive service.		
5. Check out. **Problem:** Long queue in the morning.		Use electronic key for self-checkout.
6. Reimburse expenses. **Problem:** Private services on the bill.		Split into two invoices, perhaps through TV.

use imperative language here (for example, `Find room`) to hide whether a human or a computer carries out the subtask.

The *problem part* is the only mention of the existing system. We specify problems only when they exist. For example, Subtask 2, `Record guest`, doesn't have any significant problems, so a domain-level activity description suffices. The Problem part lets us specify things we cannot specify with more traditional requirements. For instance, Subtask 1's problems demonstrate that fully automatic room allocation is inadequate.

In the right column (which we call the Support column, even though the heading is different), we outline how the new system could support the activities and solve the problems. Initially, this column shows what the system might do and is labeled "Example solutions." To emphasize the computer aspect, we write explicit subjects such as `System shows free rooms on floor map` in this column.

In a later version, the supplier might change this column to reflect new ideas or proposals, and the heading would change to "Proposal." Eventually, the column changes to reflect the services that the two parties agree to provide, and the heading changes accordingly to "Agreement."

Figure 2 shows some nontrivial solutions. For instance, some hotels might 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 such matters (`System shows bargain prices` …). Sometimes suppliers propose ideas that might exceed customer expectations. A supplier might notice that the weather influences price negotiation. Because customers might be reluctant to compare prices at several hotels during rainy weather, the supplier offers a feature for entering weather conditions. The supplier specifies this proposal in the Support column.

In some cases, solutions are trivial. Subtask 2, for instance, calls for ordinary data entry only; the analysts do not need to specify anything. Many subtasks in real systems are trivial data-entry tasks. In these cases, not much difference exists between the domain-level activity, the user activity, and the computer activity.

In industrial practice, the Problem part and the Support column are extremely useful for supporting a creative design process. At the same time, presenting the solutions in the task context limits the creativity in a cost-effective way.

High-level tasks

Until now, we have assumed that the same users will perform the tasks in both the old system and the new solution. Sometimes that assumption might be invalid, or we might plan an entirely new system without present users. In these cases, a good approach is to examine the situation from a client's perspective. 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 viewpoint, staying at the hotel is a task. It is not a traditional human–computer task because the guest and the computer do not interact directly, but the system's success depends on how well it serves guests.

Figure 3 shows the high-level guest task `Hotel stay`. Booking, check-in, and so on are tasks to the hotel team but subtasks to the guest.

We also see two new subtasks: `Select a hotel` and `Reimburse expenses`. They might help define the system's requirements. As an example, business guests need invoices to request reimbursement. The guest's `Reimburse expenses` task will be simpler if his or her personal expenses do not appear on the main invoice, and the system should support it.

High-level tasks are extremely helpful for inventing IT products, reorganizing work-

Table 1

Tracing business goals to task descriptions for a hospital system.

Business goals	User tasks							
	Dept. planner		Dept. staff			Personnel Dept.		
	1.1 Submit monthly report to Personnel Dept.	1.2 Make roster	2.1 Record actual work hours	2.2 Swap duties	2.3 Handle staff illnesses	3.1 Check rosters	3.2 Make payroll amendments	3.3 Record new employees
Personnel Dept.								
Automate some tasks	•					•	•	
Remove error sources						•	•	
Observe the 120-day rule	•					•	•	
Reduce trivial work and stress						•	•	
Hospital Dept.								
Reduce overtime pay, and so on		•						
Speed up roster planning		•						
Improve roster quality		•			•			

flow across many departments, and checking ordinary task descriptions' completeness and correctness.

A hospital case

The hotel system example is rather simple, but the techniques scale up for large, complex systems. I will show how they worked in the first complex, real-life case.

Marianne Mathiassen and I invented the Tasks & Support approach in cooperation with a large customer (a West Zealand hospital). The hospital had experienced severe problems acquiring systems through tender processes. In a tender process, the customer writes requirements and sends them out as a request for proposal. A number of suppliers send in their proposals. We studied what had happened and how it related to the requirements.

We looked at a new hospital system that recently completed the tender process. Three suppliers had submitted proposals, and the hospital had signed a contract with one of them, but the supplier had not yet delivered the system. At this point in time, we developed Tasks & Support descriptions to express requirements for the system's most difficult part: roster planning and work-hour registration. This was also the most business-critical area because the hospital management expected significant savings from improved roster planning.

The hospital had seven clear business goals for the new system. We easily traced those goals to our task descriptions (see Table 1). For example, the table's last row shows the business goal `Improve roster quality`, which means "ensure that properly qualified people are on duty at any time—for example, not two novices alone." The table shows that two tasks—`Make roster` and `Handle staff illnesses`—must support this goal. Our technique let us examine the task's details to confirm that it addressed the goal properly (by identifying it as a problem with the existing system). In contrast, the recent contract's requirements made tracing the hospital's business goals almost impossible.

What Tasks & Support covered

We compared the 38 old, feature-based roster-planning requirements to the new Tasks & Support requirements. These task descriptions (sometimes in combination with a data model) addressed 23 of the original requirements.

The task descriptions did not cover 15 original requirements. Those requirements specified that the system produce special reports. The old system could print these reports. However, nobody we talked to could explain how the hospital used the reports, so we could not identify any tasks that needed them. We believe some of those reports were related to tasks that we could have described, while others were the old system's relics.

Eight requirements were new. The task

> **Task descriptions don't cover quality requirements such as response time and usability, but they point out where quality is crucial.**

descriptions clearly showed a need for them, but the old requirements had not mentioned them. Some of these requirements could not be met by the new system recently contracted, but nobody had noticed until now. These system deficiencies prevented the hospital from achieving its main business goals. They had signed a multimillion-dollar contract for the wrong system.

We found that Tasks & Support can reveal critical requirements that traditional methods easily overlook. However, although task descriptions can cover most functional requirements, separate data descriptions are usually necessary. Task descriptions don't cover quality requirements such as response time and usability, but they point out where quality is crucial. Analysts could embed quality requirements in the task descriptions, but suppliers prefer to have them separate—preferably with references to the tasks.

Suppliers' opinions

After we completed our comparison, we asked the three suppliers to compare Tasks & Support to traditional requirements. They felt that the method had these advantages:

- It greatly clarifies the customer's needs and prospective solutions.
- It simplifies tracing between requirements and business goals.
- The supplier can specify a solution's advantages by relating it to the user tasks; the supplier can also show where its solution exceeds the customer's expectations.
- The supplier can demonstrate to the customer how the system will support the tasks and handle critical issues.
- Tasks & Support grants all suppliers equal opportunities because it does not prescribe a solution.
- The parties can adjust the solution's ambitions according to costs and needs.

They reported these disadvantages:

- Tasks & Support doesn't specify data. (A data description is needed too, as we had found out ourselves.)
- It doesn't cover quality requirements. (Suppliers prefer that they are separated from the tasks.)
- Amending the Support column is an unusual reply method; most suppliers will

need careful instructions on how to reply.
- Developers will have to work more to develop new system parts than with traditional methods. (Later experience showed that development time actually is less once developers learn to design user interfaces systematically.)
- Tasks & Support creates more work for the customer than traditional methods do. (Actually, it turned out to be much faster, as I explain later.)

About a year after we conducted our comparison, the hospital decided to use Tasks & Support to acquire a new COTS product, possibly with tailor-made extensions. The application managed cross-departmental patient administration. The hospital anticipated the contract's value at approximately \$US4 million plus approximately \$2 million annually in operating costs.

Working as a consultant, I trained two expert users and one IT specialist for two days. The users and the specialist had used traditional feature-oriented requirements, but they had never seen task or use case techniques.

Next, the two expert users worked alone to specify some of the tasks. After some initial mistakes, which I helped them correct, they completed the entire specification in 10 days. The specification work required approximately 25 client-days and three consultancy days. The work lasted three weeks. Later, the hospital team reported that similar projects used to take 25 weeks using traditional requirements.

Previously, the IT department had asked each user department (wards, labs, the personnel department, and so on) to write down their system requirements. The IT department then edited the list and sent it to each department for comments and approval. This process caused long debates about the specification's completeness and the users' needs.

Using Tasks & Support, the IT department helped a small expert-user group write a set of task descriptions, sometimes with suggested solutions. The expert users' deep task understanding was key in the approach. The team sent the Tasks & Support descriptions to the user departments for comments and approval. The departments now primarily commented on the task descriptions' complete-

ness (factual) instead of on its feature requirements (opinion). When a department suggested a solution, the analysts included it as a possible solution in the right column; that is, as an example rather than a requirement.

When the suppliers responded, they described their proposals in the right-hand column. The stakeholders reported that they had a much easier time than usual comparing the suppliers' proposals. The team spent 20 person-days comparing the two best proposals in detail. The team essentially acceptance-tested the existing product versions, working through all the task descriptions and variants to assess how well the product and the promised extensions would support them. Their comparison quickly convinced stakeholders of which supplier to choose. The traditional approach required 10 times as much work because many stakeholders had to review and comment on the proposals. The new approach also ensured that stakeholders received the task support they needed or at least knew in advance what they would not receive.

About the Author

Soren Lauesen is a professor at IT-University of Copenhagen. His research interests include human–computer interaction, requirements specification, object oriented design, quality assurance, marketing and product development, and interaction between research and industry. His industry projects have encompassed business applications, compilers, operating systems, process control, temporal databases, and software quality assurance. He is a former member of the Danish Research Council for Science and the Danish Research Council for Technical Sciences. Contact him at IT-University, Glentevej 67, DK-2400 Copenhagen NV; slauesen@itu.dk.

Reference

1. S. Lauesen and M.B. Harning, "Virtual Windows: Linking User Tasks, Data Models, and Interface Design," *IEEE Software*, vol. 18, no. 4, July/Aug. 2001, pp. 67–75.

For more information on this or any other computing topic, please visit our Digital Library at http://computer.org/publications/dlib.

Fill here?