3.6 Task descriptions

What is it?

Structured text describing user tasks.
Easy to understand for user as well as developer.
Easy to specify variants and complexity.
Simple to verify.
Domain-level requirements – also suited to COTS.

The new product must support various user tasks. Figure 3.6A shows a structured way to describe them. The description consists of several parts, one for each work area, and one for each task in the work area.

In Figure 3.6A, we have not only described the tasks; we also use them as requirements:

R1 The product shall support tasks 1.1 to 1.5.

This is a domain-level requirement. We have described the activities in the domain, and the requirement is to support these activities. The description says what human and computer should achieve together, and it doesn't mention any product features.

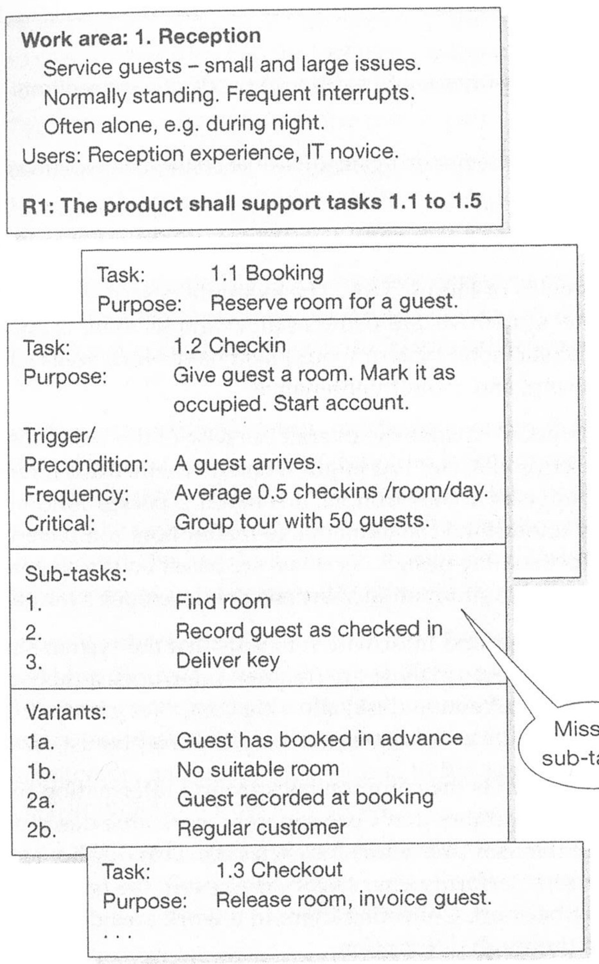
Many developers find this kind of requirement surprising. Can the requirements be verified, they may ask. Yes, we can easily check that the final product can support these activities. However, the support may be good or bad, efficient or cumbersome, so we are left with a quality comparison. We might specify the desired quality, but it is not sure we can get it. In practice, the customer will more likely compare several solutions and find the preferred one. More about that in sections 3.8 and 7.3.

If you are familiar with *use cases*, you will probably notice that task descriptions look very much like use case descriptions. What is the difference? The difference is whether we describe what humans do or what the computer does:

- A **task** is what user and product do together to achieve some goal. (The product is usually a computer system.)
- A **use case** is mostly the product's part of the task (see more below).
- A **human task** is the user's part of the task.

When the customer gets a new product, he changes the balance between what the user does and what the product does. Often, some kind of computer system was used before, and with the new product the computer system takes care of more of the task. In some cases the human part of the task disappears all together, and we have a total automation of the task.

Fig 3.6A 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.
. . .	

The term task is traditional in ergonomics and human–computer interaction, and it has focus on the human part of the total task. The term *use case* is traditional in UML and object-oriented analysis, and it focuses on the computer part of the total task as explained in section 3.12. A few authors say that use cases really are tasks, e.g. Stevens and Pooley (2000) and Wiegers (1999). Some developers use the term 'user story' to mean the same. To avoid these conflicting definitions, we use the term *task* for the combined action of human and computer.

In Figure 3.6A we describe tasks by means of a template inspired by Cockburn's *Use Cases with Goals* (Cockburn 1997, 2000).

Requirements. Task descriptions can serve several purposes in requirements:

- They can specify requirements without specifying anything about the product features.
- They can explain the purpose of traditional product requirements (explained in section 3.7).
- They can balance user demands against feasible solutions (explained in section 3.8).

Work area, background information

Let us look at the details of Figure 3.6A. The example describes the work area *reception*. In the hotel system we are using as an example, there is only this work area, but a more realistic hotel system would also have work areas such as staff scheduling, purchasing, and room maintenance.

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 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.

The work area description is the common background information for all the tasks in that work area. Most authors don't use separate work area descriptions, but give some description of the user (the actor) for each task. 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. 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 recognize 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. On Figure 3.6B the trigger is the arrival of a guest – he reports at the reception desk. In some cases there may be several triggers, for instance for the task *look at your new e-mails*, as shown on Figure 3.6B. This task has at least three triggers, and the last one is a weak trigger that occurs because "the user is curious".

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. We explain more about preconditions below.

Frequency and critical. The fields for frequency and critical are very important in practice. The requirement on Figure 3.6A 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. Next comes a 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 are on the *domain level*. They specify what the user and the computer must do together. Who does what depends on the design of the product. It is likely that the computer will help in finding free rooms, but the receptionist will make the final choice. 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.

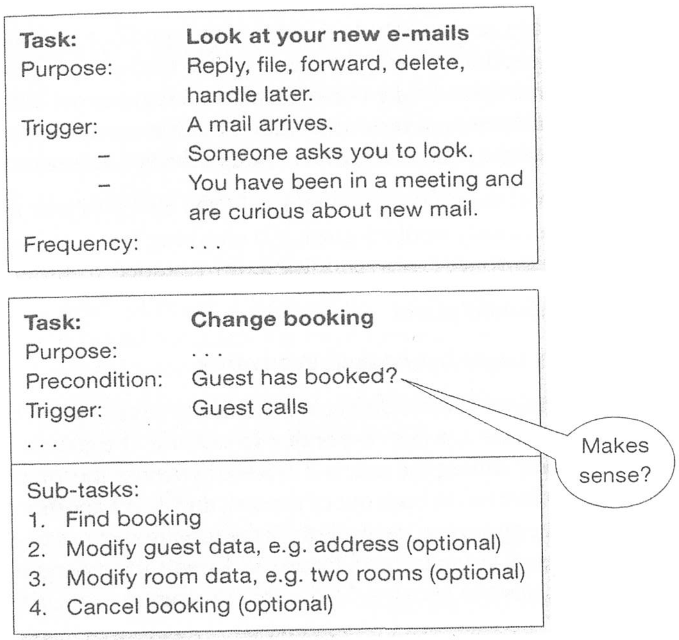
Variants are a blessing for customers as well as for developers. You don't have to describe rules or specify logic for the many special cases; simply list the variants to be dealt with. Experienced developers say that as long as there are below 20 variants, this is manageable. Above that, consider redefining the task or splitting it into several tasks.

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. In a small hotel you may, for instance, see a regular guest arriving, and since the receptionist knows him and knows that his favorite room is available, he hands him the key and later records that he has checked in. The system should allow any sequence as long as it makes sense.

Sometimes one or more sub-tasks are optional. Usually that is clear from the context, or you may specify it in the sub-task. You may also specify it as a variant of that sub-task, but that is cumbersome. As an example, let us look at the case where a guest phones to change his booking. Here we have a clear event, but it is not clear what kind of change we end up with. Maybe the guest just wants to inform us of his new address; maybe he wants to book an additional room; maybe he wants to bargain since he has found a cheaper alternative; maybe he ends up canceling the booking.

Some analysts attempt to define separate tasks for each of these possibilities, but that is cumbersome and doesn't reflect the true user situation. Figure 3.6B shows how to solve the problem by using optional sub-tasks.

Fig 3.6B Triggers, options, preconditions



Task:	**Look at your new e-mails**
Purpose:	Reply, file, forward, delete, handle later.
Trigger:	A mail arrives.
–	Someone asks you to look.
–	You have been in a meeting and are curious about new mail.
Frequency:	. . .

Task:	**Change booking**
Purpose:	. . .
Precondition:	Guest has booked?
Trigger:	Guest calls
. . .	

Sub-tasks:
1. Find booking
2. Modify guest data, e.g. address (optional)
3. Modify room data, e.g. two rooms (optional)
4. Cancel booking (optional)

Preconditions and sub-tasks

Some developers find preconditions very important. Our experience is that they may be important for use cases in order to help the programmer, but not for tasks. (There are exceptions, as discussed below.)

As an example, Figure 3.6B shows the task *change booking*. It is logical to assume that the guest actually has a booking, and the example shows it as a precondition. This makes sense if the example specified what the computer should do. The programmer would then know that some other part of the program had checked that the guest has a booking, so his program part doesn't have to check once again. Use cases focus on what the computer should do, so here we have an example of the usefulness of a precondition in a use case.

But is it really a precondition for the task? Are we sure that the guest actually has a booking? And if not, which task specifies what to do if he believed he had a booking, but actually had no booking?

The answer is simple. Sub-task 1 says what is necessary: the receptionist will find the booking and thus see whether the guest has booked. Or rather, the receptionist will find out whether the system has recorded a booking for that guest. (Maybe the guest booked on a day when the system was down, so that the booking is on a slip of paper somewhere.)

Consequently, we shouldn't specify any precondition in this case. We might specify a variant of sub-task 1, *No booking recorded*, but that is too obvious to specify.

As another example, some analysts insist that we specify a precondition for check-in, saying that the receptionist has to be logged on to the system. This makes more sense since the receptionist should get nowhere without logging on. On the other hand, this is so obvious that there is no need to mention it. Or we could claim that logon is not an essential part of the task, and that it is more relevant as a security requirement.

As a final example, assume that we had split the check-in task into two: (1) checking in a previously booked guest, (2) checking in an unbooked guest. It might then make sense to specify a precondition, for instance:

Check-in a booked guest

Precondition: Guest has booked in advance

Even this use of precondition is dubious. It somehow suggests that the receptionist knows for sure whether it is this case or not. In practice, the guest may believe that he is checked in, and only during the task is it revealed whether the 'precondition' is true. If it is not, the receptionist has to back out of the task and select another path through the system. This can be annoying, particularly if the receptionist has to enter guest data once more. For this reason, the solution in Figure 3.6A treats the difference between booked and non-booked guests as variants. Notice that the trigger event is the same in both cases.

Preconditions may be more relevant for sub-tasks in order to show restrictions on the sequence. For instance we could have described check-in in this way:

Task: Check-in

Sub-tasks:

1. Find room
2. Record guest data
3. Record check-in (precondition: free rooms selected and guest data recorded)
4. Deliver key

The precondition on *record check-in* reflects that guest data as well as one or more free rooms are needed to check in a guest. This is obvious to the user, but may be a useful hint to the developer.

In some cases a sub-task is so big that it deserves a separate task description. Essentially, task A will call task B. In this case it may be useful to have a precondition on task B stating what should have been checked before task B is started.

Extends, includes, etc.

Many analysts take great care to specify various relationships between use cases. This is very useful if we think of what the computer should do, since it helps to structure the program. The same principles are sometimes useful on the task level too. One task may, for instance:

- **include** another one as a sub-task (e.g. task A calling task B as above)
- **extend** another one, because it uses some additional variants
- be the **equivalent** to another one – only the names differ.

It can be useful to specify some of these things to avoid writing things twice and to help designers use common solutions wherever possible. However, users find them confusing, particularly if the analyst is dogmatic about the terms, which often happens. (Many developers find the terms confusing too and spend hours discussing what is what.)

Our best advice is to be pragmatic. Work on a task-oriented level that keeps the number of task descriptions at a manageable level. Use *variants* rather than a plethora of extends and includes. We have modeled large application areas by means of twelve task descriptions, each with an average of four variants. We have also seen systems of similar complexity modeled with 200 use cases on such a low level that they failed to reflect true, closed tasks.

Useful for technical systems?

Task descriptions are primarily intended as domain-level requirements for the user interface. What about the technical interfaces, for instance the interface between the hotel system and the accounting system? There are two ways to deal with the issue:

Part of a larger task. A technical interface will ultimately serve some purpose to humans, and in principle we could simply describe this user task. In the accounting case, the accountant's task is to balance the bank account, send invoices, etc. This task is an indirect requirement to the technical interface, exactly as it is an indirect requirement to the user interface. Section 5.3 explains this approach in detail.

Technical transaction. A task is a closed and meaningful piece of work for a human user. What is the equivalent concept when two technical systems (actors) communicate? It is a *transaction*. A transaction is an interchange of messages that achieves something meaningful. Maybe a task description is a good way to describe such a transaction, but I have not seen this done in practice. The advantage should be that we don't try to divide the work between the two technical systems too early. This seems promising when a main contractor tries to find suitable sub-contractors. However, if one of the systems already exists, we might better use design-level requirements. Sections 4.9 and 5.5 show ways to specify the interfaces between technical systems on the design level.

Advantages of task descriptions

Validation. Customers find it easy to validate the task descriptions because they speak the customer's language. Customers can also identify special cases to be dealt with, and the requirements engineer can deal with them immediately, simply by adding them as further sub-tasks or variants.

Trace to development. Developers can easily understand the task descriptions, but find it somewhat difficult to design the corresponding product functionality. However, once a screen design is available, it is fairly straightforward to check that it supports the task. The check can for instance be done as a walk-through of task descriptions and associated screens. An expert user should participate in this.

Verification. Checking the final system against the task descriptions is straightforward. The task description and the variants generate good system test cases. Verification is possible before the system is put into operation.

Domain-level. Task descriptions are on the domain-level and involve no product design. Task descriptions improve the intuitive understanding of the domain, omitting the need for a lot of detailed requirements.

Suitable for COTS. Task descriptions are excellent for COTS-based systems since they are supplier-independent. Different COTS products have different user dialogs, although they all support the same tasks. It is often the *quality* of their task support rather than their features that makes one COTS system better than another.

Complexity. Task descriptions can specify complexity and many variants in little space.

Disadvantages of task descriptions

No data specified. Nothing is shown about the data required for the tasks. In principle, this information is needed for developers to produce appropriate screen pictures and to check that the necessary data is available in the database. Often intuition suffices but, if not, we suggest *tasks with data* (section 3.13) to overcome the problem.

Non-task activities. Some activities may not be real work tasks, but they may still require product functionality. Examples are ad hoc reports and surfing the Web without precise goals. Section 3.10 give hints to deal with them.

Design is harder. Some developers find it difficult to translate the task descriptions into good product functions. The naive solution would be to develop special screens for each task, but this can result in too many screens, which is expensive and may also confuse the user because the data can look different in the various tasks. So, how do you get the same functionality with fewer screens? The Virtual Windows technique (section 2.5) is one answer.