User Interface Design
A Software Engineering Perspective

Soren Lauesen
You have probably noticed that some computer systems are easy to use. They have high usability. Others are a nightmare. They have low usability.

As an example, old text-processing systems were hard to use, and they couldn't even do as much as modern text processors. Modern text processors are fairly easy to use. The main difference between old and new text processors is the user interface. The functionality inside the systems is much the same, yet one interface is easy, another hard.

With a low-usability system, you wonder whether the system is stupid or you are. The general rule in this book is that only the system is stupid. We can make the systems easier to use, but we cannot change human nature.

In this chapter we explain what user interfaces are, and how we can define and measure usability.
1.1 User interface

The user interface is the part of the system that you see, hear and feel. Other parts of the system are hidden to you, for instance the database where information is stored. Although users don’t see the hidden parts, they imagine to some extent what is going on ‘behind the screen’. This imagination or understanding of what goes on is often quite important, for instance when users have to deal with a system that doesn’t work as usual.

When you use a computer, you give it orders, usually by means of the mouse and the keyboard. The computer replies, usually by showing something on the screen or making sounds. Sometimes the situation seems reversed the computer gives you instructions and you have to reply. In both cases we talk about human–computer interaction, and we call the system an interactive system. (There are also non-interactive systems. A cinema is one example – the customer doesn’t interact with the movie. A smoke detector is another example – it patiently looks for smoke all the time, but only tells the user about it when it senses some.)

The interaction with the computer takes place through the user interface. In a standard PC, the user interface consists of the screen, keyboard, mouse and loudspeaker (Figure 1.1A). In more advanced systems, the interface may include voice input through a microphone; special buttons, lights and displays; electronic gloves that can feel the movements of your fingers; and eye sensors that can detect where you are looking at the screen.

We can also talk about the user interface to a photocopier, a television or even a car. These devices actually have a hidden computer inside. In principle, we can talk about the user interface to any device, also one without any computer, such as a door or a foldable table. However, in this book we focus on user interfaces to interactive computer systems.

The computer system will often have on-line help, paper manuals, training courses, hot-line support, etc. Some people say that these are parts of the user interface too, others that they are not. Whether you call them user interfaces or not, these things are an important part of the total system as experienced by the user. For this reason it can be important to plan them in connection with the user interface.

Technical interfaces

A computer system may have other interfaces than the user interface. It can have interfaces to other computer systems, for instance an account system. It can also have interfaces to physical processes, for instance temperature sensors, valves, etc. As an example, a chemical factory is computer controlled. The computer system measures the temperature, pressure, etc., and controls the chemical process by opening and closing valves, switching heaters on and off, etc.
These technical interfaces are not user interfaces since the user doesn't interact directly across them. The user interacts indirectly with them through the user interface to the computer.

**Design of user interfaces**

In principle, it is easy to make a user interface. You just have to make it possible for the user to see and change all data in the system, and allow him to send any command across the technical interfaces.

As an example, assume that the system is dealing with sales and invoicing. It has a database of customers, products and invoices (Figure 1.1B). The user interface would just have to allow the user to create new records in the database, edit the records, print out invoices and delete old records. Modern database systems, such as Microsoft Access, have built-in screens for doing these things, and it is not necessary
to program anything to make such a system. As a system developer, you just have to
set up the necessary database tables and user windows.

Such a system has adequate functionality, but it will not be easy to use. The users
don't get the necessary overview of data for their tasks, they will have to go through
too many screens to do simple things and it is not easy to understand how to carry
out the tasks. In other words, the system has low usability.

If we insist that the system shall also be easy to use, the problem becomes so hard
that few IT-suppliers know how to deal with it. Often they get around the problem
by claiming that nobody can define what ease of use means, and even if someone
could, you cannot make the system easier to use than it is already. And even if you
could make it easier to use, it would be much too expensive to do so.

Fortunately, these IT-suppliers are wrong. You can define what ease of use means,
and you can design good user interfaces systematically. This book shows you how.
Product developers using these techniques report that it is not more costly. On the
contrary, development becomes much easier. Further, the product sells much
better.

The professional area dealing with user interfaces and usability has many names, for
instance:

  HCI: Human–computer interaction
  CHI: Computer–human interaction
  MMI: Man–machine interface

Many countries have a SIGCHI group (Special Interest Group on Computer–Human
Interaction). In the United States, we have ACM/SIGCHI (www.sigchi.org); in the
United Kingdom, British HCI Group (www.bcs-hci.org.uk); in Denmark, SIGCHI.DK
(www.sigchi.dk).

Quality factors in IT systems

From the system developer's point of view, ease of use is just one quality factor
among others. Developers talk, for instance, about:

correctness (few errors in the system)
availability (e.g. access 23 hours a day)
performance (e.g. system responds within 20 seconds)
security (e.g. preventing hacker attacks)
ease of use (often called usability)
maintainability (easy to maintain the program)

...  

The contrast to the quality factors is:

functionality (that the system has the necessary features)
Developers must deliver adequate functionality as well as adequate quality. They focus primarily on the functionality, and they are quite good at it, although they sometimes miss some features. They find it much more difficult to define and deliver the quality, so in this respect usability is not different from all the other quality factors.

Some HCI specialists claim that all quality factors are a matter of usability. Maintainability, for instance, is a matter of usability for those developers that have to correct and enhance the system.

In this book we limit ourselves to usability in daily operation of the system. We also assume that the system works correctly, is available when needed and responds fast enough. In other words, we assume that the correctness, availability and performance factors are okay. It is quite hard to obtain this, but it is not the topic of this book.
Usability. In the book we will focus on two factors that together form usability:

- Functionality. Providing the system features that allow the user to carry out his tasks.
- Ease of use. Providing the features in such a way that it is easy to learn the system and easy to carry out the tasks.

In the next section we go into detail with this definition.
1.2 Usability factors

If you ask programmers what usability means, many of them will give definitions such as the system must be menu-based with at most three menu levels; it must have on-line help; it must follow the Microsoft Windows style guide (Figure 1.2). Unfortunately, this doesn’t guarantee usability. For instance, most of us know computer systems that look like other Microsoft Windows applications, yet are almost impossible to use.

These attempts at defining usability prescribe a technical solution, but don’t catch the essentials: the user’s experience when trying to use the system. Can he find out how to use it? Does it make his work easy?

The following definition better catches the essentials.

Usability factors

Usability consists of six factors:

a) **Fit for use** (or functionality). The system can support the tasks that the user has in real life.

b) **Ease of learning.** How easy is the system to learn for various groups of users?

c) **Task efficiency.** How efficient is it for the frequent user?

d) **Ease of remembering.** How easy is it to remember for the occasional user?

e) **Subjective satisfaction.** How satisfied is the user with the system?

f) **Understandability.** How easy is it to understand what the system does? This factor is particularly important in unusual situations, for instance error situations or system failures. Only an understanding of what the system does can help the user out.

**Ease of use** (or **user friendliness**) is a combination of factors b to f.

Developers often say that it is impossible to design a system that scores high on all factors. This may be true, and in practice it may be necessary to specify the level needed for each factor. As an example, a Web-based system for attracting new customers should have emphasis on ease of learning and subjective satisfaction, while a system for air-traffic control should have emphasis on task efficiency and understandability.

In the rest of the chapter we will look at ways to measure the usability factors and specify the level needed in the final system. It is easiest to measure usability in the final system. Unfortunately, it turns out to be much too costly to improve usability at that stage. For this reason it is crucial to find ways to measure usability much earlier,
when it is still feasible to change the user interface. Fortunately, this is possible and quite cheap, as we will show throughout the book.

The usability factors above are based on Shneiderman's work (Shneiderman 1998). We have omitted his factor 'few errors', because it is covered by the factors b, c, d and f above. In return, we have added the factor 'understandability'. Traditionally, usability experts were reluctant to talk about the user's understanding. You couldn't readily measure it, and what really mattered was whether the user could perform his tasks. Today, understandability is more recognized. It is crucial for mastering complex systems with many functions, and we can measure understandability in the same way that we give marks at exams.

Game program usability

For some systems, the usability factors above are dubious. One example is game programs. They need not be easy to learn or task efficient – it is no fun if you too easily beat the opponent. So what are the important usability factors? The most relevant one is the subjective satisfaction – how much the users like the game program. But in order to improve a game program that users don't like, we need to dig deeper into the subjective satisfaction.

Rouse (2001) has identified some additional usability factors for games, for instance that games must be entertaining, challenging, fair, and for some, games allow socializing. These factors are important for games, but irrelevant for systems that support user tasks.

Importance of usability

Who is responsible for the usability of the final system? Programmers and other developers take responsibility for the technical correctness of the system and for the necessary functionality (fit for use). But who is responsible for the ease-of-use factors?

Traditionally, it was the user department's responsibility. They had to write user documentation and train users. That was the only way to compensate for poor usability in these traditional systems.

This is slowly changing. Mature development teams feel it is their joint responsibility that the system is easy to use. Although they feel a responsibility, they are still grappling with how to deal with it.

Why is usability more important now than earlier? Because the IT technology has become cheaper and cheaper, while users have become more and more precious. Usability pays in many ways: users save time, more people can use the system, and people can handle many computer systems and don't have to specialize in a single system. Furthermore, the technology allows us to do it. The main barriers are that
Fig 1.2 What is usability?

Max three menu levels: ??
- On-line help
- Windows standard

Responsibility?
Programmers?
Other developers?
User department?

Usability factors:
- a. Fit for use (adequate functionality)
- b. Ease of use
- c. Task efficiency
- d. Ease of remembering
- e. Subjective satisfaction
- f. Understandability

Measurable
Priorities vary

Game programs:
- a. ??

developers don’t know how to achieve usability and they fear it is too costly. Fortunately, both barriers can be overcome.
1.3 Usability problems

A usability problem is anything that hampers the user, for instance that he cannot figure out how to carry out his task or finds it too cumbersome. In order to improve the usability of the system, it is important to identify the usability problems.

Usability problems are a special kind of system defects. The system works as intended by the developer, yet the user interface is inconvenient or hard to understand. Let us look at the major kinds of defects.

Defect types

- **Program error** (or bug). If the system doesn't work as intended by the programmer, we have a program error – a bug. Examples are that the system crashes or shows something wrong on the screen.

- **Missing functionality.** If it is impossible to carry out the task, the system is not fit for use. We classify this as a usability defect – missing functionality.

- **Ease-of-use problem.** If the system works as intended by the programmer and it can support the task, yet the user cannot figure out how to do it or doesn't like the system, the system is not easy to use. This is another kind of usability problem. We classify them according to their severity to the user, for instance a task failure or a minor problem. More on that below.

Here are a few examples of usability problems (Figure 1.3).

**P1:** The user cannot figure out how to start the search. The screen says that he should use F10, but for some reason he doesn't see it until he has tried several other ways.

**P2:** The user believes that he has completed the task and that the result is saved, but actually he should have pressed Update before closing the window.

**P3:** The user cannot figure out which discount code to give the customer, although he knows which field to use.

**P4:** The user says that it is completely crazy having to go through six screens in order to fill in ten fields.

**P5:** The user wants to print out the list of discount codes in order to put his own notes on it. After many attempts, he calls hot-line. They say that the system cannot do that.

Usually we classify usability problems according to their severity to the user. Here is a classification we often use.
Problem classification

- **Missing functionality.** The system cannot support the user’s task. Problem P5 is of this kind.
- **Task failure.** The user cannot complete the task on his own or he erroneously believes that it is completed. Problems P2 and P3 are of this kind.
- **Annoying.** The user complains that the system is annoying or cumbersome; or we observe that the user doesn’t work in the optimal way. Problem P4 belongs here.
- **Medium problem.** The user finds the solution after lengthy attempts. P1 is this kind.
- **Minor problem.** The user finds the solution after a few short attempts. P1 would have been this kind if the user had found the solution fast.

**Problems versus observations.** It often happens that one user gets stuck on a certain problem, while another user soon finds the solution. Is the problem then a task failure? The answer is that severity is not a classification of the usability problems, but of the observations (occurrences) of the problems. Problem P1, for instance, might be observed twice. When user A encountered it, it was a task failure; when user B encountered it, it was a minor problem. If many users have task failures for P1, we might call P1 a task failure, but some judgement is involved to do this.
In practice we want to correct the most serious problems, the critical problems.

**Critical problem**

- Missing functionality for an important task, or
- Task failure that occurs for many users, or
- Annoying to many users.

We may also try to correct less serious problems if we can easily do so. Usually, we cannot ensure that all users succeed with everything - even though it would be wonderful. It may be impossible to do so, or it may be too costly.
1.4 Basics of usability testing

The most effective technique to find the usability problems is a usability test. There are many variants of usability tests. Below we outline our favoured, low-cost, high-effective technique. The description should be sufficient for your own first attempt at usability testing. When you have tried it once on your own, you will appreciate Chapter 13, which gives you more detail and practical advice.

Think-aloud test. During a usability test, we let a user (the test subject or test user) try to carry out realistic tasks using the real system or a mock-up of it (Figure 1.4). There are several ways to do this. Our favoured technique is to ask the user to think aloud — explain what he is doing and why.

Real system. You may want to find usability problems in a system that is finished or at least working to a large extent. The user will use the system to carry out various tasks.

Prototypes and mock-ups. Early during the design process, there is no real system to test with. However, you can find usability problems with a prototype of the planned system. The simplest prototypes are mock-ups. They may consist entirely of screens on paper, and the user would fill in the fields by means of a pencil, 'click' on buttons with the pencil, etc. The leader of the test (the facilitator) would take paper windows away when they 'close' and bring them up when they open. Mock-ups are very useful early in development because they are fast to make and easy to throw away. More advanced prototypes run on the computer and look right, but have little functionality. (More on prototypes and testing in Chapter 2.)

Test team. It is best to have two or three persons on the test team: The facilitator talks with the user, the log keeper notes down what happens, in particular the problems the user encounters. A possible third person can observe how the test proceeds and help the other two as the need arises.

Plan the test

Test users. When you plan a usability test, you have to find test users. Choose people that might be typical users. If we are developing a Web site that will help people find public transportation from one point to another, the test users should be ordinary people with little IT knowledge. IT people will not be good test users since they know too much about what goes on behind the screen. They will not notice problems that would stop ordinary people.

If we are developing a system that will help hotel receptionists do their job, we should use test users with some reception background. Again IT specialists will not be good because they won’t notice some problems that would stop non-IT people, while they may encounter problems relating to hotel terminology that wouldn’t bother the receptionist.
Test tasks. You also have to choose some test tasks or situations where the user will use the system. As an example, if we want to test a system for hotel receptions, a good test task will be to book a room for a guest. There are many other tasks that should be tested too, of course. Choosing the right test tasks is a critical issue if we want to find all usability problems. A good test task should meet these criteria:

- Something the user would do in a real work situation. Booking a room for John Simpson is good, but booking one for Donald Duck is childish and suggests to the receptionist that the system is not for professional use. Changing the system set-up is not a realistic task for a receptionist.

- A full piece of meaningful work (a closed task). Booking is good – the receptionist handled the customer’s call. Logging in is not a good task. The receptionist has not accomplished anything by logging in.

- Stated without hidden help – without hints on how to carry out the task.

Hidden help is a matter of how we explain the task to the user. If we are testing a Web site for finding public transportation, don’t use a task stated like this:

Wrong: Find a bus connection around 11 p.m. from route 6, stop 12 to route 8, stop 23.

This might be the way the Web site asks for start and end of the trip, but users would not state their problem in this way. You might try a more vivid scenario instead:

Better: You are planning to go to a party tomorrow in 20 Brickwood Street, Brighton. You would like not driving home to 55 Westbank Terrace, Richmond. Is there any public transportation that could help you? How late? And what would it cost?

This also makes it possible for the test user to imagine the situation and draw on his intuition.

Study the system. Finally, you also have to learn how to use the system yourself. Otherwise you cannot understand what the user attempts and where he misunderstood something. If you are a member of the development team, this is easy. If not, you must experiment with the system on your own or have someone to guide you.

Carry out the test

When a test user arrives to the usability test, he is a bit nervous. He is scared of looking stupid. For this reason it is important to explain the purpose of the test:

We want to find out where the system is hard to understand or inconvenient. We know the system too well, so we cannot see it ourselves. We need your help. If you have problems with the system, it is the system’s fault – not yours.
Fig 1.4 Usability test – think aloud

**Purpose:**
Find usability problems

![Diagram of a usability test session]

1. **Facilitator**
   - Listens
   - Asks as needed
2. **Log keeper**
   - Listens
   - Records problems
3. **User**
   - Performs tasks
   - Thinks aloud

**Plan**
- Test users:
- Test-tasks:
- Study system yourself

**Carry out**
- Explain purpose:
  - Find problems when using the system
  - System's fault – not yours
- Give task – think aloud, please
- Observe, listen, note down
- Ask cautiously:
  - What are you looking for?
  - Why...?
- Help users out when they are surely lost

**Reporting**
- List the usability problems – within 12 h

Talking a bit about daily matters and the user's background also helps the user relax.

Next, you give the user the first test task and ask him to carry it out by means of the system. Invite him to *think aloud* by explaining what he does and why.

Then observe what the user does and listen to him. Write down brief notes on what happens, particularly when something caused problems.
When you cannot understand what the user is doing or why he gets stuck, ask him what he is looking for, and why he did as he did. Don't tell him how to do it until he has had lots of time to find out for himself.

When the user is truly lost and seems unable to succeed on his own, help him out with a hint. In this way he can help you observe areas in other parts of the system. However, when you help him out, you have to record it as a task failure – the user couldn't complete the task on his own.

If you cannot scribble your notes fast enough, it is okay to ask the user to wait a moment.

Reporting

After the test, you have to write a list of the problems the user encountered. They must be in such a form that people knowing the system understand what you write. You must write this list as soon as possible, preferably within 12 hours. Otherwise you cannot any longer understand your own scribbling and remember what happened during the test.

See Chapter 13 for how to write notes during the test and how to report the problems.
Heuristic evaluation and user reviews

Usability testing may sound cumbersome. Aren't there other ways to identify the usability problems? Maybe we could let a usability specialist look at the screens and point out the problems? This approach is called **heuristic evaluation**. We could also let an expert user look at the screens and discuss them with him. This approach is often called **user review**.

**Heuristic evaluation**

Heuristic evaluation can be made in many ways:

- We can let usability specialists look at the screens, or we can make fellow developers look at them.

- They can use their sound judgement and earlier experience, or they can use a list of **heuristic guidelines**. Guidelines may, for instance, say that the screens should not be too crowded, the error messages should tell the user what to do, there must be features for undoing, etc.

- Each evaluator may deliver his own list of defects, or we may ask all of them to reach agreement and come up with a common list.

Unfortunately, in many cases heuristic evaluation finds lots of problems, but about half of them are false in the sense that they don’t cause problems to real users. It would be a waste of time trying to correct these false problems, but we don’t know which of the problems are false. Furthermore, in these cases heuristic evaluation misses about half of the severe problems that real users encounter (Figure 1.5 illustrates this). I have jokingly called this the first law of usability:

**First law of usability**

Heuristic evaluation has only 50% hit-rate.

Would it help to have more usability specialists look at the screens? Yes, we get a few more hits, but a lot more false problems.

The first law is quite controversial, and many HCI specialists consider heuristic evaluation the best technique. My colleagues and I have often observed the low hit-rate, and several authors report similar results (e.g., Bailey et al. 1992; Desurvire et al. 1992; Cuomo and Bowen 1994). In section 2.3 we look at an example. It is a test of an early prototype, and the heuristic evaluators are fellow developers. In such cases the first law works well. If you carry out exercises 2.2 and 2.3, you will make an.
early prototype of a system, usability test it, and make a heuristic evaluation. You can then see whether the law worked in your specific case.

Although I have called it a law of usability, it doesn’t work in all cases. Section 14.4 shows a precise comparison of usability testing and heuristic evaluation for a finished Web site. The evaluators were among the best HCI people in the world. In this case there were no statistically significant differences in hit-rates, but heuristic evaluators rated the problems as more serious than usability testers.

Although heuristic evaluation is quite popular, we strongly warn against it as the primary approach – particularly for early prototypes. It may supplement a usability test, but cannot replace it. If you get a heuristic evaluation of a system you design, there will be some problems that you immediately recognize as things you didn’t think about. Work on them, but ignore for the time being problems that you feel are false. With a usability test it is different: you cannot reject problems as false. Some user actually encountered this problem. Chapter 14 explains much more about heuristic evaluation.

User review

A review is normally made with expert users who know the application area very well. Go through the screens with the expert user, show how they are to be used, and discuss how various tasks are carried out, what is missing, what is cumbersome, etc.

Such reviews are important for finding missing functionality. The expert user can point to many things the system cannot do. However, reviews are not good for finding ease-of-use problems. The expert user typically knows too much to encounter all the ease-of-use problems that would block typical users.

Would it be better to review with ordinary users? Unfortunately not. The way most designers conduct the review hides ease-of-use problems. The designer explains how it works and the user doesn’t need to figure it out for himself. We have often seen designers present a new system to a group of users and managers. The designer shows how it works and everybody finds it great. But if you let the users try it on their own just a few minutes later, they may not be able to use the system at all.

Conclusion

- Review with expert users to find missing functionality.
- Use heuristic evaluation with caution. Correct the problems you feel are real, and let usability tests decide for the rest.
- Always make usability tests to find the true problems.
Fig 1.5  Heuristic evaluation

Purpose: Find usability problems
Usability specialist looks at system using common sense and/or guidelines
The specialist lists problems (Consults with other experts)

First law of usability:
Heuristic evaluation has only 50% hit-rate

- Predicted problems
- False problems
- Actual problems
- Missed problems
1.6 Usability measurements and requirements

**Usability factors.** In section 1.2 we looked at usability factors, for instance ease of learning and task efficiency. The factors are our dimensions in the usability world. We might, for instance, want to go in the ease-of-learning direction and the task efficiency direction at the same time.

**Measurements.** If we want to say how far we are in these directions, we must be able to measure the dimensions, but the usability factors don’t say how to do this. Counting the usability problems is one way to measure the factor ease of learning, but there are other ways, as we will see in this section.

**Requirements and targets.** Once we know how to measure the dimensions, we may set requirements. We might, for instance, say that we want a maximum of 9 usability problems. This is a requirement and the figure 9 is our target.

In this section we will look at ways to measure usability. This can be important in many situations:

- When we develop a new system, it is practically impossible to correct all usability problems. Usability measurements can help us determine when the system is good enough.
- We may want to buy a system for a certain purpose. There may be several vendors of such systems, and as part of our comparison, it is useful to measure the usability of each system.
- We may want to expand an existing system that has been in service for some time. It is a good idea to measure its usability to see whether usability should be improved at the same time.

For each of the ways to measure usability, we will look at how useful it is during development and which usability factors we can measure with it. There are also standards in the area, see ISO 13407 (1989), ISO 9241 (1998) and ISO/IEC 9126 (2001–2002).

1.6.1 Task time (performance measurement)

The classical way to measure usability is to measure the time users take to carry out various tasks. This is also called performance measurement. Figure 1.6A shows two examples.

**ATM example**

The first example specifies usability measurement for an ATM (Automatic Teller Machine). Notice that the specification consists of three parts:
Fig 16A Measuring usability – task time (performance)

<table>
<thead>
<tr>
<th>ATM:</th>
<th>Users: 20 bank customers, random selection.</th>
<th>How to measure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 1:</td>
<td>Withdraw $100 from ATM. No instructions.</td>
<td>What to measure</td>
</tr>
<tr>
<td>Measure:</td>
<td>How many succeeded in 2 min?</td>
<td>Requirement – target</td>
</tr>
<tr>
<td>Task 2:</td>
<td>Withdraw as much as possible ($174).</td>
<td></td>
</tr>
<tr>
<td>Measure:</td>
<td>How many succeeded in 5 min?</td>
<td></td>
</tr>
<tr>
<td>Range:</td>
<td>Task 1: 18 succeed.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Task 2: 12 succeed.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Internal ordering system</th>
<th>Users: 5 secretaries in the company.</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 1:</td>
<td>Order two boxes of letter paper +...</td>
<td>What to measure</td>
</tr>
<tr>
<td>Measure:</td>
<td>Average time per user.</td>
<td>Risky!</td>
</tr>
<tr>
<td>Range:</td>
<td>Average time below 5 min.</td>
<td></td>
</tr>
</tbody>
</table>

Pros: Classic approach, Good when buying.
Cons: Not good for development; Not possible early. Little feedback.

1 How to carry out the measurement
2 What to measure
3 The requirement or target

How to carry out the measurement. In the example we have specified that we at random select 20 customers visiting the bank, and ask them to participate in a usability test. They are given two tasks: to withdraw a standard amount of money, and to withdraw as much as possible from the account.

Sometimes it may be useful to specify a bit more on how to carry out the measurement, for instance that we hand the users a credit card, that we don’t give them any instructions on how to use the ATM. We may also specify how to select the users at random.

What to measure. Basically, we measure how much time the users take to carry out the tasks. In order to end up with a single number, we then count the number of users doing the job in 2 minutes for the first task, and in 5 minutes for the second task. There are of course many other ways of ending up with a few numbers.

Requirement. Finally, we have specified what is good enough – the requirement. Our target is that 18 out of 20 users complete the first task within 2 minutes, and 12 out of 20 users complete the second within 5 minutes.

1.6 Usability measurements and requirements 23
From where have we got all these figures – 18 out of 20, 5 minutes, and so on? A lot of judgement is involved, as we will discuss in section 1.6.2.

Order system example

The second example specifies usability for an internal ordering system in a large company. The secretaries use the system to order stationery for use in their department. This is something they do rarely, probably around once a month.

The specification follows the same pattern. Notice that we talk about test users who have not been using the system for some time.

In this example we try to reduce the number of measurements to a single result by taking the average of the task times. However, this would in practice turn out to be a bad idea. Why? Because some users may fail to complete the task. We cannot include them in the average, but excluding them would hide that many users had serious problems. One way to get around it is to use two measurements: the average of those users completing the task, and the number of users that failed.

Usability factors

Which usability factors can we cover with task time measurements? Let us look at them one by one:

a) Fit for use. We can, in principle, test that all user tasks can be supported. In practice this is hard for two reasons: we may not be aware of all tasks, and it may be overwhelming to include all of them in a usability test.

b) Ease of learning. Task time measurement is well suited for this. The ATM example shows how.

c) Task efficiency. This usability factor is about efficiency for experienced, daily users. Task time measurement is well suited for this too. We just have to specify that we will use experienced users for the test. However, when we develop a new system, we have a problem: we don’t have experienced users yet.

d) Ease of remembering. Task time measurement is in principle suited for this too. The order system example shows how. When developing a new system, we have a problem because we have no suitable users yet. Some HCI specialists run another usability test with the same users a few days later. This will, to some extent, show how easy the system is to remember. In a typical stressed project, however, this is rarely done.

e) Subjective satisfaction. Task time measurement is not suited for this. Experience shows that there is little correlation between task time and the user’s subjective satisfaction with the system.
Understandability. Task time is not suited for measuring ease of understanding. We might try things like asking users what they believe this error message means, and measure their time to produce a correct answer. But this is not a good way to measure understanding.

Task times: pros and cons

Task times can measure many usability factors. It is also the classical way of measuring usability.

When we want to buy a system, we can run usability tests with it to see how fast our users can learn to perform their tasks. Usually, the system is already used somewhere else, so we might even get access to experienced users and measure how fast they perform various tasks.

If we develop a new system, task time measurements are possible in principle, but less suited in practice. First of all, you need a working version of the system to measure task times. This is only possible late in development, and at that time it is very expensive to change the system to improve user performance. Second, we don’t have access to experienced users, so we cannot measure task efficiency or ease of remembering.

The worst problem is probably that we get too little feedback for developers. We cannot see the usability problems after a task time measurement. If we know that only 10 out of 20 users completed the task, what is wrong? How can we improve?

Asking the test user to think aloud is an excellent way to identify and understand the usability problems. However, this slows the user down, so that the task times become too long.

Finally, selecting the right test tasks is a critical issue no matter whether we develop a new system or buy an existing one. For some systems it is hard to find meaningful test tasks at all. For instance, this is the case with entertainment products.

1.6.2 Choosing the numbers

We have more or less arbitrarily chosen the numbers in the examples. Why have we chosen 20 test users in the first example and 5 in the second? Why have we chosen 2 minutes as the target – the acceptable task time? Why do we insist on 18 out of 20?

The short answer is that it is a subjective assessment of cost versus benefit, combined with experience about what is possible (Figure 1.6B).

As an example, testing with 20 users takes more time than testing with 5. On the other hand, we have a good chance of finding rare usability problems also. The fact is that some problems in the user interface block almost every user, while other problems block only few of the users. We have a good chance of finding these rare problems if we test with 20 users, but not if we test with 5.
If the bank wants 90% of its customers to succeed with the ATM, it has to look also for the rare problems – those that block only 10% of the customers. So it should spend the cost of testing with around 20 users to have a good chance of including some of those 10% in the test. In contrast, the internal order system is not that crucial. The users can get help from their colleagues or the hot-line if they don’t succeed, and so it is not so important to find also the rare problems. On the other hand, testing with just two users may let too many semi-frequent problems undetected. We return to this issue in section 13.4.

As another example, why have we chosen 18 out of 20? It doesn’t make sense to ask for 20 out of 20. There will always be some users that for various reasons have troubles. On the other hand, asking for 10 out of 20 would mean that every second user had troubles. Most likely, that would harm our business because rumours would spread that our ATMs were hard to use. So, the 18 out of 20 matches a goal of 90% success. (Because of the randomness of our sample, we cannot be sure that 90% of the entire user population will succeed although 18 of the test persons succeeded. We will not discuss these statistical issues here.)

Finally, why have we chosen 2 minutes as the critical task time for the ATM? This might be based on observations of what people typically do with present ATMs. The new one should not be worse, but insisting on making it vastly better might be risky and costly. If the example was about the first ATM ever produced, we couldn’t use this argument, but had to imagine an ideal way for the users to learn it – or observe users in similar situations elsewhere, for instance using a vending machine.

Actually, tests such as in the example were carried out with the first ATMs. Reaching the stated targets was very hard at that time. Today two things have changed: the ATM user interfaces have improved immensely and users have got more experience with such devices.

Open target. In practice, there is a way around choosing the targets early on. Leave the targets open and later see what is possible. It may turn out that we can get more than we expected or that we were too optimistic. However, we have specified what kind of thing we are after and how we want to measure it.

This approach is called open target. A good way is to combine it with our expectations. We could, for instance, state the ATM example with an open target:

18 out of 20 users must complete the task within ___ minutes. (We expect around 2 minutes.)

See Lauesen (2002) for more about open targets and cost/benefit of quality in general.

1.6.3 Problem counts

Instead of measuring task times, we can use think-aloud tests and list the usability problems. This gives better feedback to developers, and can be done early in development with a prototype of the user interface.
Figure 1.6C shows how we can specify usability in this way. The specification follows the same pattern as for task time measurements.

**How to carry out the measurement.** In the example we run usability tests with three potential users. They all carry out the same tasks – thinking aloud – and we record a list of all the problems they encounter. For each problem we also record the users who encountered it, and how severe it was to each user.

In principle, we could identify the usability problems in other ways, for instance through heuristic evaluation. Because of the first law of usability, we don’t recommend this approach.

**What to measure.** In the example, we count the number of users encountering a critical problem (task failure, annoying, etc.). We also count the number of medium problems on the problem list. Now, what is a medium problem? Basically we record whether it was of medium severity to one or more users. We may choose to call the problem ‘medium’ if it was of medium severity to two or more users.

**Requirement.** The target is that at most one user will encounter a critical problem. In other words, two of the three users will be able to carry out all the tasks on their own. We also require that there are at most 5 medium problems on the list.

We can count the problems in many other ways. The figure is just an example.

**Number of users.** Are three users really enough? In principle, we should have more – about 10, but testing becomes more expensive. In practice, we measure usability
several times during development, correcting usability problems after each round of
testing. The first time, one user is enough – we will find many serious problems with
a single user, and testing with more at this stage is largely a waste of time. In the next
rounds of testing, we may use three users and get more detailed results. In this way
the total number of users will become larger than three. In section 13.4 we look closer
at the number of users.

Usability factors

Which usability factors can we cover with problem counts? Let us look at them one
by one:

a) **Fit for use.** The choice of tasks is critical, just as for task time measurements.

b) **Ease of learning.** Problem counting easily deals with this.

c) **Task efficiency.** Problem counting deals only indirectly with this factor. If the
user complains that the system is cumbersome (one kind of critical problem),
we know that there is a task efficiency problem. Users with experience from
similar systems are more likely to complain. Even if the user doesn’t notice an
efficiency problem, we may observe that he didn’t follow the fast way through
the system.

d) **Ease of remembering.** Problem counting is in principle suited for this. We
simply specify how many problems second-time users may encounter. This
works fine if we are buying a new system where we can find second-time users.
When developing a new system, we have the same problem as for task time
measurement: we have no second-time users. We can, however, run two usability
tests with the same users a few days apart. Comparing the problem counts will
give a good indication of the ease of remembering.

e) **Subjective satisfaction.** Problem counting may give some indication of this
because user’s comments during thinking aloud often reflect their opinion of the
system as a whole. However, it is hard to get a real measurement of the
satisfaction in this way.

f) **Understandability.** The thinking-aloud tests show a lot about the
understandability, but cannot give a real measurement of it.

Some HCI specialists work with a further usability factor called **few errors.** Our
problem counts would be a direct measurement of this factor.

**Game programs.** Measuring the usability for game programs is hard because there
are no clear tasks. However, problem counts are useful for specifying that the users
shall be able to operate the game as intended by the designers. In order to specify the
entertainment value of the system and other usability factors specific for games, we
cannot use problem counts of course. Opinion polls are better suited for this (see
section 1.6.5).
Problem counts: pros and cons

Problem counts are best for measuring ease of learning, but can give good indications for the other usability factors.

The great advantage is that usability can be measured early during system development. The usability tests can be done based on a paper prototype. No functional system is necessary. The usability test is at the same time a natural part of good development, and it gives excellent feedback to developers about what to correct.

Selecting the right test tasks is a critical issue, just as for task time measurements.

1.6.4 Keystroke counts

Keystroke counts can, to some extent, help us specify and measure task efficiency for experienced users. Figure 1.6D shows a way to do it in the ATM example.

**How to carry out the measurement.** We specify one or more user tasks. In practice, we specify only tasks where task efficiency is critical. In the example, we have specified the frequent case where the user just wants to withdraw a standard amount and has done it several times before. (Think of all the other customers waiting behind him in the queue at lunch time, and you can understand why this may be a critical task.)

**What to measure.** We count the number of keystrokes, mouse clicks, and other operations that the user would have to make to complete each task.

**Requirement.** We specify the maximum number of keystrokes, mouse clicks, etc., for the system to be good enough. In the example the requirement is at most 6 keystrokes including the PIN code.
To help us calculate the total task time, we have also specified the total response time for the system. The total response time is calculated in this way: the time the system takes to respond to the first keystroke, plus the time to respond to the second one, and so on. It will thus include times for reading the card, checking that the pin code is okay, checking that the account has enough money, counting the bills and dispensing them, and printing a receipt. According to the requirements, the total must be at most 8 seconds on average.

Calculating the task time

In principle, we may not really care about the number of keystrokes. The only thing of interest to us is the total task time. We can calculate the task time from the figures above. First we have to measure how long a time an experienced user on average takes for a keystroke. This figure is about 0.6 seconds for typical ATM keys because they are heavy to press. (For ordinary keyboard keys, the figure is around 0.2 seconds.) We can then calculate the total task time in this way:

\[
\begin{array}{l}
6 \text{ keystrokes} @ 0.6 \text{ seconds each} & 3.6 \text{ seconds} \\
\text{Total system response time} & 8.0 \text{ seconds} \\
\hline
\text{Total task time} & 11.6 \text{ seconds}
\end{array}
\]

This calculation is a bit too simple. First, since the user takes 0.6 seconds to push a key, any system response time needed during that period doesn’t count.

Second, the user may have to read messages on the screen and understand them. We have assumed that the user is so experienced that he doesn’t have to do this.

Finally, what is the task time really? Above we have assumed that the task starts when the user pushes the first key, and terminates when the system has dispensed the money and printed the receipt. The customers waiting behind in the queue don’t consider this the true task time. They include the time the user fiddles with his card, getting it out of his wallet and inserting it into the machine, and the time he fiddles with receipt, money, and the card to get them back into the wallet.

Since many ATMs have no shelf to put all of these things, fiddling may take much longer than the computer transaction. It should be mentioned that many ATMs don’t have a shelf in order to avoid users leaving their stuff there. The developers have made a design compromise between avoiding a disaster (forgetting the stuff) and reducing task time.

When we talk about PC systems for daily activities, the situation is somewhat different. The task time here is closely related to pushing keyboard keys, moving and clicking the mouse, waiting for the computer to respond, reading the screen, and sometimes talking to a customer meanwhile. Card et al. (1980) have shown how to compute task times for such systems based on keystroke counts (the keystroke-level model). They have also carefully measured times for pushing a key, moving the
Keystroke counts: pros and cons

Keystroke counts are a good way to specify task efficiency for short, simple tasks. The big advantage is that early in development we can see whether the requirement is met. We don’t even need access to real users.

One disadvantage is that we cannot be sure that real users find out how to do it in this efficient way. We can, however, get some indication of it through usability tests.

Another disadvantage is that keystroke counts can give us a false understanding of what the task really comprises. It is usually much more than pushing keys, as shown in the ATM example above.
1.6.5 Opinion poll

Opinion polls are the classical way to measure the user’s satisfaction. We ask a number of users to complete a questionnaire, such as the one outlined in Figure 1.6E.

How to carry out the measurement. With this technique, we ask users about their opinion, typically with questionnaires using a Likert scale. The example shows just three steps on the scale: agree, neutral, disagree. It is customary to have more steps, for instance agree, partly agree, neutral, partly disagree, disagree.

We can ask about the user’s opinion on many matters, for instance the system is easy to use; the system was easy to learn; the system helps me do things faster; it is fun to use the system; it speeds up my tasks; I will recommend it to people I know.

Typically, we would ask many users to complete the questionnaire. This is of course possible only when the system has been used widely for some time. During system development, it is customary to let each test user fill in a questionnaire after the usability test. This doesn’t give as much data, of course, because we have relatively few test users.

What to measure. Getting from the questionnaires to the measurement is easy. We just count the number of people that have marked each of the boxes.

Requirement. There are many ways to specify the target. On the figure, we have specified that 80% agree that the system is easy to learn, and 50% agree that they will recommend it to others. Instead of this, we could take averages, count worst cases, etc.

SUMI. Some usability specialists use the SUMI approach for the questionnaires. SUMI has 50 questions you can ask for any system. It also has a standard set of weights for these questions, so it ends up with one single number to characterize the usability. Furthermore, it has a database of how other systems fare on this scale so that you can compare your own system against others (see http://www.ucc.ie/hfrg/questionnaires/sumi/).

Usability factors

Which usability factors can we cover with opinion polls? It seems that they could cover all factors – we just have to ask the right questions. But let us have a closer look:

a) Fit for use. Opinion polls have a great advantage here. We don’t have to specify the tasks; we can let the users state whether their tasks are well supported. We can even ask them to write on the form which tasks are not well supported. In that way we can learn what is missing. Of course, this works only for systems that have been used for some time. Even here, users are quite often unaware of the missing task support, so we cannot get complete data.

b) Ease of learning. At first sight it seems that opinion polls can give us reliable data on ease of learning, even if we use it after a usability test early during
development. Unfortunately, users often say that the system is easy to learn although they have spent a surprising amount of time learning the system. They even say so if they haven’t learned to use it properly. So there is little correlation between the user’s subjective opinion and more objective measurements.

c) **Task efficiency.** Here too, opinion polls seem a good measurement. And again, users often express satisfaction with their system in spite of evidence that the system is inconvenient and wastes a lot of user time. (If the boss knew about this, he would not be as satisfied as the users.)

d) **Ease of remembering.** Here too, opinion polls seem a good measurement. I must admit that I don’t know to what extent the opinion matches objective measurements.

e) **Subjective satisfaction.** Opinion polls should be strong here. We simply ask the users whether they are satisfied. If they say yes, we trust them. It is their subjective opinion we are after, even if it has nothing to do with objective measures of task time, etc. Alas, people don’t say what they really mean. Marketing people and psychologists have known this for many years. For that reason they ask the same thing in many ways to get down to the truth. We have a bit of it in the example in Figure 1.6E: ‘Would you recommend the system to others?’ In principle, it is the same as asking whether they like the system themselves. If they say they do, but don’t mean it, they will usually not recommend it to others. Asking whether they think their colleague likes it, is a trick of the same kind.
Cultural differences are involved too. Asians typically say that things are good – feeling that if not, it is their fault. Europeans are different. They may complain about everything, and are more frank in their opinions.

f) **Understandability.** This too can be measured by opinion polls, but I don’t know how well it correlates with objective measurements.

**Game programs**. Opinion polls are probably the best way to measure the special usability factors for game programs, for instance the entertainment value, the challenge, the fairness, etc.

**Opinion polls: pros and cons**

Many HCI specialists consider opinion polls the best measure of usability. They stress that opinion polls give consistent results from user to user. This may be true, but as explained above, it doesn’t mean that the results match objective evidence. The second law of usability is at work:

**Second law of usability**

There is little correlation between subjective satisfaction and objective performance.

Subjective satisfaction is heavily influenced by organizational factors, for instance management style and good lunch breaks. Developers cannot control these factors.

Another problem with opinion polls is that they are hard to measure during development. We get more reliable results when we poll after delivering the system. If results show that users are satisfied, we cannot be sure that the system really is easy to use. If results show that users are dissatisfied, we don’t know what the problem is, but it is a sign that developers should go and talk to the users to find out what the real problems are (see Henderson *et al.* 1995; Frøkjær *et al.* 2000).

1.6.6 **Score for understanding**

If we want to measure whether the system is easy to understand, a good approach is to ask users how they believe the system works. Figure 1.6f shows an example.

**How to carry out the measurement.** We ask users a number of questions about the system’s behaviour. In the example we ask what various error messages mean when using an ATM.

We could also ask questions of the kind ‘What do you think the system would do if . . . ?’ In the ATM case we could, for instance, ask ‘What do you think the system would do if the wire connecting it to the bank was broken?’ Or ‘What do you think
What to measure. We assess how correct the answers are, for instance as exam marks on the scale from A to D. This is a somewhat subjective assessment of the correctness, but gives a good indication anyway. (Remember that it is not the user we mark, but the system's understandability.)

Requirements. In the example, the target is that 80% of the answers (across all users) get marks A or B.

When I try the first ATM question with people, I get very few correct answers. Few people know that the message 'amount too large' may indicate that although they have ample money in their account, they have exceeded their daily cash allowance or their monthly limit. Both limits are there to prevent robberies and fraud, but most users are unaware of this.

Score for understanding: pros and cons

An understandability test can give us information that is hard to get with task time measurements or problem counts. It is particularly useful for finding usability problems in error messages. The reason is that today's systems have a huge number of error messages. Trying tasks where the user can encounter many of them is extremely time-consuming. Testing as shown in Figure 1.6F is an easy way out.

A further advantage is that we can test understandability early in development. When we test error messages, the test also helps us improve the message. This can even be done late in the project, where developers come up with most of the error messages. If we want to buy an existing system, we can carry out the test before we buy or sign the contract.
The main disadvantage of a score for understanding is that it measures only the understandability factor.

1.6.7 Guideline adherence

In many cases it is an advantage that the user interface follows some recommended guideline, for instance the style guide for Microsoft Windows applications. The guideline says things such as:

- Each window must have a cross in the top right corner. Clicking on the cross must close the window.
- The active window must have a blue title bar.
- Short-cut keys must be provided for all system functions in order to allow mouse-free operation.
- An example of a screen that follows the rules is shown in...

We can measure how well the system follows such a guideline. Figure 1.6G shows an example.

*How to carry out the measurement.* In the example, we ask an independent expert to scrutinize the user interface and identify the deviations from the guideline. Since there often is doubt about what a guideline actually means, it might be better to have two experts look at the interface and agree on the significant deviations from the guideline.

*What to measure.* Since we can expect more deviations in large systems than in smaller ones, we count the average number of deviations per screen. (A screen might be a window or a Web page.)

*Requirement.* In the example, the target is at most one deviation per screen. In practice, this is quite a strict requirement. In general, developers find it difficult to follow guidelines, even if they try hard.

Usability factors

Let us see how such a requirement contributes to the usability factors:

a) **Fit for use.** The guideline cannot say much about the specific user tasks to be supported. However, it may remind developers to support generic tasks, for instance printing, importing, and exporting data.

b) **Ease of learning.** Following a guideline definitely helps users learn a new system faster. Of course it works best if the users already are familiar with other software following the guideline. But even if they are not, the guideline may help
create consistency across various parts of the new system, thus making it easier for users to move between system screens.

Following a guideline does not ensure that the system is easy to learn. The guideline can only help learning the superficial aspects of the system.

c) **Task efficiency.** Following a guideline does not ensure that the system is efficient for daily use. The guideline may remind developers to provide short-cut keys, etc., but often this is not sufficient for effectiveness. In some cases a guideline may even prevent highly efficient solutions. As an example, the required use of message boxes and warnings in Microsoft Windows slows down experienced users. Designers are becoming aware of this, which is why you see more and more message boxes allowing you to click ‘don’t show this warning again’. Unfortunately, even experienced users hesitate to click this box, because they feel uncertain of the consequences.

d) **Ease of remembering.** A guideline can strongly help users remember how various things are to be done, because it is the same as in other parts of the system.

e) **Subjective satisfaction.** Following a guideline used in popular systems probably makes people more satisfied with the new system.

f) **Understandability.** A guideline helps users understand what the system does, but only on a superficial level. The guideline cannot help users understand the application-specific issues that relate to the world outside the computer system.

**Guideline adherence: pros and cons**

In general, it is an advantage to follow a guideline, particularly if it is also used in other systems that users know. Following a specific guideline cannot in itself guarantee high usability. In some cases, a guideline may hamper high task efficiency.
Experience shows that it is quite hard for developers to follow guidelines, particularly if they consist of dozens of pages. What works best are guidelines built on examples. Developers will then find an example corresponding to what they want to do, and copy the necessary parts to their own system.

Some consultants help large companies develop a more specific guideline for their applications. They do so by carefully designing the user interface for an existing company application and usability testing it. This example is then the core of the company's own guideline. The advantage is that it can provide guidelines in company-specific issues that a public guideline cannot. Another advantage is that it is based on an example, which developers find much easier to follow.

1.6.8 Which usability measure to choose?

Above we have looked at many ways of measuring usability. They all have their strengths and weaknesses. Which one should we use in practice? The short answer is that we have to combine several measures. Which ones depend on the kind of project and on the important usability factors for that system. Figure 1.6H summarizes what the different measurement techniques cover. The dark boxes show where the technique is particularly good.

If we talk about developing systems for non-critical daily use, ease of learning and understandability are the most important factors. My favourite choice in this case is problem counts + score for understanding

Figure 1.6H shows that these techniques cover ease of learning and understandability, and that the techniques are also suited early during development. If we don't develop such a system, but buy it, we should use task time measurements instead of problem counts. In this way we better cover task efficiency.

If we talk about developing a Web system to attract customers, subjective satisfaction becomes important. I would add opinion polls and use many test subjects carefully selected to cover the many kinds of users we expect to attract. If we talk about developing a system that must support time-critical tasks, I would instead add keystroke counts for the critical tasks.

As far as I know, ease of remembering is rarely measured. We somehow assume that ease of learning also provides ease of remembering. Whether this is true, remains to be tested.

In conclusion, no two projects are alike – consider the proper choice of measurements for each project. In large industrial projects you may have to use different combinations of usability measures for different user groups. See Lauesen (2002, section 10.4) for an example.
Test yourself

a) Mention some examples of quality factors.

b) Mention some examples of usability factors.

c) Mention some examples of usability requirements.

d) What is the difference between a factor and a requirement?

e) What is a usability problem?

f) What is a critical problem?

g) What is the difference between usability testing, heuristic evaluation, and user review?

h) Mention two good ways to measure ease of learning.

i) Which of them are good when you buy a finished system and when you are developing a new system?

j) What is an open target requirement?

k) What do the first and the second law of usability say?

Design exercises. See Chapter 17.