Published in Howard et al. (eds.): Human-Computer Interaction
Interact'97, Chapman & Hall, 1997, pp. 15-22.

Usability Engineering in Industrial Practice

Søren Lauesen

Copenhagen Business School, Howitzvej 60
DK-2000 Frederiksberg, Denmark
slauesen @ cbs.dk

ABSTRACT Do developers use proven usability techniques like user involvement, usability testing, and iterative design in industrial practice? Based on inside knowledge of many different types of projects, the author must conclude that these techniques are seldom used. There are several reasons for that. For instance:

(1) The market pressure is not there. Users ask primarily for functionality, and cannot formulate their usability requirements.

(2) Developers misunderstand the usability techniques. For instance they assume that usability testing is a kind of debugging, rather than a step in designing the interface. Or they believe that expensive labs have to be used, rather than a low-cost approach that can be learned in a day and carried out by developers.

(3) There is no proven way to make a good, first prototype. Since development in practice seems restricted to modifications of the first prototype, it is essential to make it good.

(4) There is no proven way to correct observed problems.

KEYWORDS usability engineering, software engineering, industry, iterative design, thinking-aloud

1. INTRODUCTION

Most system development is dealing with rather traditional systems for business, banking, government, technical purposes, embedded systems, etc. How are user interfaces designed in such systems? Do developers use proven techniques like user involvement, usability testing, and iterative design?

It is not easy to get a true picture of the situation through questionnaires because experience shows that developers often misunderstand terms like usability testing and iterative design. Also, they often say they do something, when observation shows that they do not.

This report is based on my own observations. I have inside knowledge of many development projects in Denmark. With a background as a developer, I am often consulted in usability issues, give courses in interface design, and talk a lot with people from industry. I have observed many kinds of projects like standard business administration (accounting, invoicing, etc.); shipyard administration; hospital support; teacher administration; TV broadcast planning; insurance applications; network software packages; process control systems; embedded systems, etc.

So what is the general picture? Do developers care about usability? Yes. Do they do something about it? Very little. Do they know what to do? Partially, but often they do it incorrectly, misunderstand the approaches, or give up after incomplete attempts.

In this paper I will explain what typically goes wrong and what seems to work. Basically, I will try to look at usability issues from the developer's perspective, comparing them with the technical side of development.

I will concentrate on the proven techniques: user involvement, usability testing, and iterative design. Other techniques are important but not yet available in a form ready for general industrial use. One example is task analysis which covers many things and overlaps with software engineering concepts like "use cases", "scenarios", and requirements specifications. See Lim (1996) for a discussion of the industrial applicability of task analysis.

2. USER INVOLVEMENT

It is a proven approach to involve users during the entire development process from analysis to deployment. Grudin (1991) discusses the many obstacles to user involvement, so I will just report how my Danish observations fit into his picture.

The degree of user involvement seems to depend on whether the project has a specific customer or aims at a particular market:

Specific customer

These projects can be in-house projects, for instance in banks and insurance companies; or they can be software house projects for a specific customer. In these cases we always find a great deal of user involvement during analysis and design. Particularly with in-house

projects it is common to have users in the project team during these phases. However, the "Scandinavian Model" where users actually formulate the system objectives and do part of the design is very rare in Denmark.

The users involved are "expert users" with a good understanding of the application domain. Management gives their involvement in the new system a relatively high priority, but the pressure of many daily activities can make it somewhat difficult to find time for cooperation. However, the amount of user involvement seems somewhat better than what Grudin (1991) describes.

User involvement during usability testing, however, is not as successful; as will be discussed later.

Product for a market

Products can be business applications, technical systems, or embedded software. In these projects we see all the problems mentioned by Grudin: Marketing tries to represent the users, barriers are set up to prevent developers from contacting users, the international market is difficult to contact "face to face", etc.

3. USABILITY TESTING

Usability testing is probably the most important step in assuring usability. Without a usability test we can only make guesses at the level of usability. In order to understand why developers don't readily accept usability testing, it may be useful to compare usability testing with traditional program testing.

When a developer tests a program, he tests whether it works correctly with the hardware and other technical environment. Usability testing does not test the correctness of the program, but whether the user can work correctly and conveniently with it. During a usability test, the system - or a prototype of it - is used by typical users to carry out realistic tasks that the system was intended to support. The testers observe the users and write down any problems encountered by them. Next, developers try to repair the problems, and a new usability test is made to check whether the problems have disappeared. The process will typically have to be repeated several times, just as you have to repeat program testing.

Developers are confident with technical program testing. They have learnt to accept that they commit errors and that most errors can be identified through testing. They have also learnt how to locate the source of the error and how to correct it, and they know that repeated testing is necessary to make sure that the error actually disappeared. Their managers know that testing is costly and so provide good test equipment to save working hours.

Not so with usability testing. Developers are usually surprised that users can have problems with a program that is technically correct, and they doubt that it is the developer's responsibility to fix such problems. On several occasions I have heard product developers discuss what the market value of increased usability would be. They tend to conclude that sales wouldn't increase due to increased usability. In some cases users have complained about usability, and that seems to have some effect. Mostly, however, users complain about functionality and compare products based on functionality. So that is where efforts for the next release are directed.

Managers seem to share many of these beliefs and are reluctant to fund usability testing. The critical "test equipment" for usability testing is the users. Developers will normally have to find users themselves, sometimes fighting with marketing and managers to be allowed to contact users.

If developers try to correct the usability problems, they soon realize that it is difficult to find ways to do it. The reason for this is that while they know how the technical environment works in painful detail, they don't know how the users "work" or think, and consequently they cannot find good solutions.

While a developer wouldn't dream of shipping a program without careful testing, most programs are actually shipped without any kind of usability testing.

In spite of all the motivational issues, some project teams try to do some usability testing. They encounter many procedural problems that may cause usability testing to fail. I will discuss some of them.

3.1 When to Test?

Ideally the usability tests should be made early in development, in order to allow a complete redesign. Early prototypes are suitable for that. In practice, however, this is almost never done. Usability testing is started late in the development cycle, almost as a kind of debugging or technical program testing. This issue is discussed further in Section 4.

3.2 Use Video and Usability Lab?

Many developers believe that usability testing requires a usability lab with video and computer log of the user's actions (Neal & Simons, 1984). Therefore they conclude that they cannot afford usability testing since they have no access to such a lab. This is a very unfortunate misunderstanding, since most usability testing can be done fast and efficiently without a lab. Even many usability experts believe that a lab is necessary. In Denmark this misunderstanding probably arose because a large software house invested in a lab and used it to market themselves as usability conscious. In many cases this seemed to override earlier attempts to promote a low-cost technique in Denmark (Jørgensen, 1990).

Large American companies also put much emphasis on usability labs, and often start with the lab as a promotional thing (Wiklund , 1994). I have even heard usability specialists complain that to management, the usability goal is "to utilize the lab 80% of the time".

In Section 5, I have outlined a low-cost technique used extensively by my colleagues and myself for usability testing. The main feature is to manually make a log of the problems as they occur during the test, and write a final problem list soon after the test, based on the log and memory. This is quite reliable in most cases and

saves the expensive equipment and a lot of work viewing the video tapes.

Our technique finds ease-of-learning problems, and also *soft performance problems*, where the user finds the system too slow or requiring too many user actions.

Hard performance problems, where fast reactions or motoric aspects are the issue, require eye movement sensors and video to identify detailed task steps, fast and unconscious user reactions, etc. Examples are support systems for pilots or surgeons. However, there are few systems of this kind.

3.3 Thinking Aloud or Observe Only?

Some proponents of usability testing stress that it is important not to interfere with the user. They recommend a silent test where the user's actions are logged and video taped while he works in his normal fashion. Later the tester studies the actions in detail. This approach may be correct if we just want to observe the problems, but most usability testing deals with finding problems *and* correcting them.

To observe the usability problems is one thing, to find their cause and repair the problems is a different issue. In order to do that, it is essential to know what the users actually thought, and why they did not do this and that. This information is not available from the user's behavior or a video tape. The easiest way is to have the users think aloud during the test, or cautiously ask them what they are trying to do in case their behavior seems strange.

The very act of thinking aloud causes the users to proceed differently, for instance with a different performance, but do they encounter different problems? Henderson et al. (1995) and Hoc & Leplat (1983) have investigated this issue. They compared four techniques:

1. Thinking-aloud: The users think aloud while doing the task
2. Record and thinking-aloud: User actions are recorded and later replayed while the users explain why they did what they did.
3. Record and study: User actions are recorded and later studied by the testers.
4. Explain later: The users try to do the task and later comment on the problems they encountered.

Technique 1 and 2 seem to reveal the same problems, but technique 2 is much more time consuming. Techniques 3 and 4 don't reveal the correct problems. Technique 3 is also very time consuming.

I would add that techniques 1 and 2 reveal soft performance problems. Hard performance problems, where fast reactions or motoric aspects are the issue, can be revealed by technique 2 or 3.

The recommendation for all ordinary applications is to use only technique 1: Thinking aloud.

3.4 Which Test Tasks?

Developers initially have some difficulties defining good test tasks. Here is a typical badly defined task from a system that can measure and compare sounds and noises:

Task: Set sensor sensitivity
1. Open settings window
2. Select the sensor
3. Set the sensitivity to 10 mW

Testing with this task will not find out whether the user is able to perform a real-life task. One weakness is that the task description is a step-by-step instruction that tells the user how to perform the task. But we wanted to find out whether he could do it on his own. Another weakness is that the task is not closed and meaningful to the user. The user would never use the system just to set the sensor sensitivity. He wants to make a measurement, and setting the sensor sensitivity is a necessary step that he might not think of himself. We also want to test that he discovers this step.

A better task would be:

Task: Compare the noise at the top and bottom of the dishwasher
Equipment: Sensor supplied in a separate box. Dishwasher is a dummy during prototype testing (a small table will do)

This task is closed and meaningful, and we don't provide "hidden assistance". We also leave it to the user to find the proper sensitivity - just as in real life. My experience is that when developers have seen a few examples of good and bad tasks, they can define good tasks on their own.

3.5 Developer in the Test Team?

Some usability specialists make usability tests without developers on the test team. The developers get the report later and may see the video tape of the session. The reason is that the developer would interfere with the test, discuss issues with the users, or guide the users.

This is a realistic risk, but there is a serious drawback of the approach: It drastically reduces the chance of having the problems corrected. First of all, developers seem to distrust a report stating various problems that they have not experienced themselves and cannot reproduce. (The same pattern is seen when unrepeatable technical errors are reported to developers.) Second, they cannot see what the real problem is, why the user did not see what was at the bottom of the screen, etc. As a result they fail to make a proper correction of the system. If they had been present during the test, they could have asked such questions.

In my experience, many developers are very good participants in a usability experiment once they get the idea of the whole thing. With a bit of support during the first session, they find a good balance between inappropriate interference and important information gathering. Jørgensen (1990) reported similar success with test

teams consisting of just one developer.

3.6 Rely On Standards and Heuristic Evaluation?

Some developers believe that adherence to standards or various kinds of heuristic evaluation (e.g. design inspection) will ensure usability. It would be nice if this were true, but at present these techniques cannot replace usability tests.

Standards (or style guides) improve learnability for users knowing other systems that follow this standard. However, domain-specific problems and many other problems cannot be covered by a standard. For instance, no standard can specify what terms to use for domain-specific concepts. Only a usability test can reveal whether the developer got it right. Several studies have shown that a check against standards only find about 25% of the problems users encounter, although they find a lot of standard violations that users don't notice (Cuomo & Bowen, 1994; Desurvire et al., 1992; Jeffries et al.,1991).

In some cases, we have observed usability problems caused by a standard. A good example is the use of modal dialogue boxes under MS-Windows. Many users complain that in order to enter the data needed in the dialogue box, they have to see the windows behind the dialogue box. But they cannot move windows around or bring other windows forward until they have closed the dialogue box.

Heuristic evaluation can be an expert's inspection of the design, or a check against guidelines. Surprisingly, heuristic evaluation finds only about half of the problems that users encounter (Cuomo & Bowen, 1994; Desurvire et al., 1992; Jeffries et al.,1991). Furthermore, about half of the problems reported with heuristic evaluation are "false" in the sense that real users do not notice these problems. Trying to remedy the false problems is a waste of development effort. Generally, heuristic evaluation should only be used to detect the most obvious problems. If a problem seems dubious or is difficult to repair, let the usability test reveal whether it is important.

In conclusion, standards and heuristic evaluation may help, but do not eliminate the need for a usability test.

3.7 Usability Test Or Demo?

Talking with developers, I often hear them say that they made a usability test of their latest product. Great, I say, how many problems did you find? None, they say, the users were very happy with the system. That surprises me a bit, since we usually find 20 to 30 problems during a one hour usability test. A closer discussion reveals that what the developers did was a demo of the system. They showed the system to the users, walking through typical cases. The users were invited to comment on the system, but did not notice any problems.

In my experience, users can find a system very attractive when seeing a demo of it, yet be completely unable to perform anything with it on their own. The system is thus not easy to learn.

I have noticed several times that developers feel very uncomfortable with the thought of a real usability test, whereas they love to make demos. The reason may be their pride in their own work combined with the suspicion that the users will just mess up the whole thing. (This is what we actually expect users to do during the usability test. That is why we make it).

In early stages of development, a demo may be very useful for finding missing functionality, but that has little to do with usability factors like ease-of-learning.

3.8 How To Find Test Subjects?

Maybe the greatest obstacle in usability testing is to find test subjects. I have observed several times that this issue blocks serious thoughts about usability testing. The responsibility for finding test subjects is usually with the developers, but they hesitate and don't know how to go about it. They have few social contacts with users, and their attitude is far from approaching users or customers whom they don't know. In many product developing companies they are not even allowed to. Grudin (1991) discusses many other barriers between users and developers. Sales and marketing staff have much better potential for getting user contact. I strongly suggest that support in finding test users is planned early and provided from other parts of the company.

Since test subjects are difficult to find, the same test subjects are often used for testing successive system versions. The result is that the new version seems surprisingly much better than the old version. The reason may be that the users learned the concepts in the first version and successfully transferred the concepts to the next version. New users would likely encounter more problems.

A related mistake is to test the system with users who have been involved with analysis or design of the system. They also know too much to be representative users. However, as pointed out by Carlshamre & Karlsson (1996), expert users are very good at finding missing functionality as part of a usability test.

If we are to test the ease-of-learning aspect of the system, we should use new users for testing each new version. Testing for performance, however, assumes experienced users, and the same users may well test several versions if we allow them to gain experience with each version.

4. ITERATIVE DESIGN

As mentioned earlier, the first usability tests should be made on an early prototype in order to allow a complete redesign. In practice, however, this is almost never done. Usability testing comes in late, almost as a kind of debugging or technical program testing.

In order to understand why it is so, it is useful to compare the iterative design with traditional program development. Experienced developers proceed in this

way:

Design: The overall structure of the system is developed. The process is highly iterative, particularly in the first part of design. The product of design can be inspected by fellow developers, but it is not in a form where it can be tested.

Programming: The different modules are programmed. It may happen that the design has to be modified, but a complete redesign is considered a disaster. Modifying what you have programmed is always the preferred solution.

Testing: The modules are tested individually and in combination. Many errors are detected, but mostly they can be repaired through simple program modifications. Reprogramming a module or redesigning a part is considered a sign of bad craftsmanship.

It is obvious that developers try to use the same pattern on the user interface. They assume that they can conceive a good design, inspect it (heuristic evaluation), implement it, and then correct the problems. But in reality, we cannot inspect whether or not the interface design is good, we have to test the design, as discussed under heuristic evaluation above.

Even when an early prototype is made, it turns out to be so difficult to conceive a new design that it is never done. What we have conceived seems so precious that we prefer to repair it, rather than start all over. Bailey (1993) has made the same observation in an experimental setting, and he concludes that the first prototype is *the* major factor in how usable the system can become.

4.1 Problem Correction

A late usability test is better than no usability test. When a test has been made and the problem list is available, it is time to remove the problems. But this is another point where the process often stops. There are several reasons for this:

1. The long list of problems was a surprise. Actually, the development team had expected that the test proved that the user interface was good. And they have no time to remove the problems or cannot see how they could remove them.
2. The developers have some time to remove problems, but cannot agree with the "customer" on what to remove.
3. The developers try to remove some problems, but the problems don't seem to disappear.

4.2 Problem Classification

Even if there is very little time available, some errors can be removed. The question is: which errors? Many development teams try to prioritize the problems, but cannot agree on the priorities. The problem is that they unconsciously mix up the importance to the user with the difficulty of removing the problem. Separating the two factors helps. I have good experience with this approach:

1. **User importance.** On the list of problems, note the number of users that encountered the problem. Also classify each problem according to its importance to the user. For instance, use a scale with these steps: **failure** (task failure), **slow** (performance problem), **inconvenient** (small performance problem), **difficult** (to learn), **minor** (fast to learn). When classifying the problems it is important not to worry about how to correct the problem, because it influences your judgment of the importance to the user.
2. **Difficulty.** Let the problem list rest in the mind at least one night. That often brings ideas for solution. Then classify each problem according to how difficult it is to repair. A simple scale could be: **small** effort (change of screen text, etc.), **medium** effort (several modules to be changed), **large** effort (new logic and data structure), **uncertain** how to repair.
3. **Cost/benefit.** Now prioritize according to user importance compared with difficulty of repair (cost/benefit). This can be formalized, but a discussion based on the two factors is usually sufficient.

4.3 Removing a Problem

In programming, it is usually obvious how to remove an error once we have found its cause. The repair may be more or less costly, but the solution is usually clear. In usability issues it is not that easy. We can roughly distinguish these classes of problems:

1. **Obvious solution.** It is easy to say what the proper solution should be, although the actual repair can be small, medium, or large. This is the frequent case in programming, but in my experience less than half of the usability problems are of this kind.
2. **Wrong solution.** You believe you have a solution, but actually the solution does not work. This is frequent in usability, but rare in programming. Note that only a new usability test can reveal whether the problem was of class 1 or 2.
3. **Unknown solution.** We understand the problem and its cause, but don't have any idea how it could be overcome. This is frequent in usability, but extremely rare in programming.
4. **Unknown cause.** We can observe the problem, but cannot find its cause. The think-aloud approach in usability testing can almost eliminate this class of problems. In programming it often requires hard work to find the cause of the problem, but once found, the solution tends to be obvious.

Unfortunately the only thing to do about cases 2 and 3 above is to experiment with various solutions. In some teams I have worked in, we have had to realize that we could not find a solution within the existing framework. The problem could be, for instance, that the users did

not have a consistent domain terminology, so whatever terms we used, some users misunderstood them. We had to rely on a written user introduction or personal instruction to overcome the problem. (This is of course better than not knowing the existence of the problem.)

It would be wonderful to have a usability handbook where you could look up various problems and find a set of possible solutions. However, this is an area for further research.

5. A LOW-COST USABILITY TEST TECHNIQUE

Below I have summarized a usability test technique used extensively by my colleagues and myself. It is easy to learn: we practice one session with developers and help them write the problem list. Then they seem able to do it on their own.

1. Test team: At least one developer and preferably another person knowing the system and the test technique. If the system is a prototype, the developer is the Wizard of Oz, simulating the system. If there is another person, he keeps a manual log during the test, focusing on the problems encountered by the users. If not, the developer keeps the log.
2. Preparation: Make sure that the system is in the right state, data base initialized, etc. Have copies available of windows with blank fields. Have a tape recorder and tapes ready (optional).
3. Start of test: Explain the test purpose to the user, the system purpose and the domain, but don't give more introduction to the system than would realistically be given in real life. (Many developers assume that users will have a course on how to use the system, but in real-life users rarely get such courses, or they get them only after having already used the system for some time.) Start the tape recording (optional).
4. Give the users exercises with real-life tasks. State the purpose of the task, but don't explain how to perform the task. Give them only one exercise at a time.
5. Ask the users to think aloud while trying to perform the tasks. If testing with two users at a time, encourage them to discuss what to do and why.
6. During the test, log the rough flow of the dialogue (menu points selected, windows brought up). When a user encounters a problem or makes a comment on the system, make a note in the log. If convenient, use blank copies of screens to sketch the situation. Log also the clock at various points of the session and the point where you change tapes.
7. If the users asks for help, encourage them to find the solution themselves.
8. If the users have searched for the solution for some time, help them out, but make sure that you have made a note of the problem. Such a problem is to be categorized as a Task Failure, i.e. that the users couldn't complete the task on their own. So make sure that the users had ample time to try on their own. By experience, the developer finds ten seconds a long time, while the users finds it a brief moment.
9. If you don't understand what the users are doing, ask cautiously.
10. End of test: Terminate the test itself after roughly one hour. Everybody will be tired at that time. Interview the users about what they liked and what they didn't like. To get an impression of their understanding of the system, ask them what they believe the system would do in some specific cases of your choice.
11. Within 24 hours, write a list of the problems detected. This list may include problems from two or more test sessions. Unless you have ample time, don't listen to the entire tape recording, but refer to it in cases where you are not quite sure what happened. Classify the problems as described in Section 4.2.

A usability test session should last about 1.5 hours in total. Usually, we will run two or three sessions and then write a joint problem list. It takes about 2 hours to write the problem list. The report will generally contain 20 to 30 problems of various severity.

On a number of occasions, we have had two independent log keepers. Each log keeper wrote their own problem list. Later we compared them. The agreement was quite good: 90% of the problems were in both lists.

CONCLUSION

Observations show that proven usability techniques are used very little in industrial practice. We can summarize the causes in this way:

Barriers:

Developers are concerned about usability, but doubt that much can be done about it. They also doubt that it is worth doing anything, or that it is their responsibility.

The market pressure is not there. Users ask primarily for functionality, and cannot formulate their usability requirements.

Developers have difficulties finding users, particularly for usability testing. When seeing the results of a usability test, developers are surprised at the number of problems, and don't know how to correct them.

Misunderstandings:

Developers assume that usability testing is a kind of debugging, rather than a step in designing the interface. Or they assume that a demo of the system is a usability test.

Developers often believe that expensive labs have to be used, rather than a low-cost approach that can be learned in a day and carried out by developers.

Developers believe that standards and inspection/evaluation are sufficient.

Where HCI theory falls short:

There is no proven way to make a good, first prototype. Since development in practice seems restricted to

modifications of the first prototype, it is essential to make it good.

There is no proven method for correcting observed problems.

REFERENCES

Bailey, G. (1993): Iterative methodology and designer training in human-computer interface design. Proceedings of InterCHI'93, pp. 198-205.

Carlshamre, P. & Karlsson, J. (1996): A usability-oriented approach to requirements engineering. Proceedings of ICRE'96, pp. 145-152.

Cuomo, D.L. & Bowen, C.D. (1994): Understanding usability issues addressed by three user-system interface evaluation techniques. Interacting with Computers, Vol.6, No.1, pp. 86-108.

Desurvire, H.W., Kondziela, J.M & Atwood, M.E. (1992): What is gained and lost when using evaluation methods other than empirical testing. Proceedings of HCI 92, pp. 89-102. Cambridge University Press.

Grudin, J. (1991): Systematic sources of suboptimal interface design in large product development organizations. Human-Computer Interaction, Vol.6, pp. 147-196.

Henderson, R. Podd, J., Smith. M. & Varela-Alvarez, H. (1995): An examination of four user-based software evaluation methods. Interacting with Computers, Vol.7, No.4, pp. 412-432.

Hoc, J.M. & Leplat, J. (1983): Evaluation of different modalities of verbalization in a sorting task. Int. J. Man-Machine Studies, Vol. 18, pp. 283-306.

Jeffries, R., Miller, J.R., Wharton, C., and Uyeda, K.M. (1991): User interface evaluation in the real world: A comparison of four techniques. CHI'91 Proceedings, pp. 119-124. ACM 0-89791-383-3/91/0004/0119..0124.

Jørgensen, A.H. (1990): Thinking-aloud in user interface design: a method promoting cognitive ergonomics. Ergonomics, 1990, Vol. 33, No.4, pp. 501-507.

Lim, K.Y. (1996): Structured task analysis: an instantiation of the MUSE method for usability engineering. Interacting with Computers, Vol.8, No.1, pp. 31-50.

Neal, A.S. & Simons, R.M. (1984): Playback: A method for evaluating the usability of software and its documentation. IBM Systems Journal, Vol 23, No 1, pp. 82-96.

Wiklund, M.E. (ed.): Usability in practice. AP Professional, 1994.