Published in: Proceedings of REFSQ'98, Presses Universitaires de Namur, 1988

Six Styles for Usability Requirements

Soren Lauesen & Houman Younessi

slauesen@cbs.dk, HYounessi@swin.edu.au
Copenhagen Business School, Howitzvej 60
DK-2000 Frederiksberg

Abstract. A system can have adequate functionality, but inadequate usability because it is too difficult to use. The purpose of usability requirements is to guard against that. This paper shows six styles for usability requirements seen in practice or recommended by experts. For each style we discuss how we can verify the requirements, how we can use them during development, how we elicit the data for the specification, and to what extent the style covers the essence of usability.

Introduction

The largest part of the requirements specification deals with the functional requirements, that is the system input, processing, and output. These requirements say nothing about how easy the system is to use, yet ease-of-use is a major concern with most systems.

A usability requirement specifies how easy the system must be to use. Usability is a non-functional requirement, because in its essence it doesn't specify parts of the system functionality, only how that functionality is to be perceived by the user, for instance how easy it must be to learn and how efficient it must be for carrying out user tasks.

Surprisingly, the literature has very little to say about usability requirements and rarely provides real-life examples. Nielsen (1993), Preece (1994, chapter 19), and Macaulay (1996) give much advise on usability requirements, but in a rather abstract setting without real-life examples.

Also practitioners have great difficulties specifying usability requirements and often end up stating that "the system shall be easy to use". Every now and then, however, we have come across more meaningful and precise usability requirements. They have been important for this study.

The usability requirements must be tangible so that we are able to verify them and trace them during the development. They must also be complete so that if we fulfill them, we are sure that we get the usability we intend. Meeting these goals is difficult in practice and no approach seems to cover all of them. Below we study six usability styles and discuss their strengths and weaknesses. In practice you can use the styles in combination.

Usability Factors

Before we look at the different styles, we will briefly discuss what usability is. According to a traditional definition, usability consists of five *usability factors*:

1. Ease of learning. The system must be easy to learn for both novices and users with experience from similar systems.
2. Task efficiency. The system must be efficient for the frequent user.

3. Ease of remembering. The system must be easy to remember for the casual user.
4. Understandability: The user must understand what the system does.
5. Subjective satisfaction. The user must feel satisfied with the system.

The different styles specify and measure these factors more or less directly.

Developers often say that it is impossible to make a system that scores high on all factors. This may be true, and one purpose of the usability requirements is to specify the necessary level for each factor.

Usability Testing

An important fact, confirmed by many experiments, is that *nobody can foresee the usability problems for a given user interface* - not even usability experts. Usability experts may predict many usability problems with a design, but about half of the predicted problems are false, in the sense that users don't feel they are problems. What is worse, the usability experts miss about half of the problems that real users experience (Cuomo & Bowen, 1994; Desurvire et al., 1992; Jeffries et al.,1991). Only some kind of testing with real users can reveal the usability problems. In order to correct the problems, we need to identify them early during development. As a result, usability

Example1.1: Performance-based usability spec of an ATM

This is an example of a performance-based usability specification for an automatic teller machine (an ATM). There are five enumerated requirements (R1.1 to R1.5). To help developers understand them, they are justified by higher-level goals.

. . .

It must be easy to learn how to use the ATM. Otherwise we cannot expect customers to switch to our system. In particular, success with the first attempt at withdrawing cash is important. Attracting new customers is also important.

R1.1 Customers with ATM experience from other banks: In their first attempt, they must be able to withdraw a preset amount of cash within an average of 2 minutes.

R1.2 Customers without previous ATM experience: In their first attempt, 90% of them must be able to withdraw a preset amount of cash within 4 minutes.

R1.3 Customers having tried to withdraw a preset amount: In their first attempt, 70% of them must be able to withdraw a non-preset amount within 6 minutes.

To reduce waiting time when many customers queue up to withdraw cash, the performance for experienced users is important too.

R1.4 Customers with at least 6 withdrawals over at least a month: They must be able to withdraw a preset amount of cash within an average of 30 seconds.

To reduce customer annoyance and need for personal help from staff, it is important that customers understand the causes of malfunctions.

R1.5 When the system rejects a transaction: In 90% of such cases, the customer must be able to explain in his own terms what the cause is and what he can do about it. Examples: it is a transmission problem (I should try again later), I used a wrong PIN code, I have overdrawn the account, I have exceeded my daily allowance, the card has expired.

specifications that cannot be tested during development have a serious weakness.

The kind of testing needed to reveal usability problems is called *usability testing*. It is an experiment where real users with relevant background try to perform real tasks by means of the system or a prototype. Observers record the problems encountered by the user and the time to perform the tasks.

1. Performance style

In a performance-based usability specification we define a set of tasks where usability is important, we define one or more user groups, and we define performance objectives for the user groups when performing these tasks.

Example 1.1 shows a performance-based usability specification for an ATM. It is based on three tasks:

1. Withdraw a preset amount of cash chosen by the user from a short list of choices.
2. Withdraw another amount.
3. Handle transaction rejections (trouble shooting). This is not a separate task, but a common variation of other tasks.

The specification mentions three user groups:

1. Users with other ATM experience
2. Users without ATM experience
3. Routine users of the new ATM

Performance objectives for group 1 and 2 specify something about ease of learning, while performance objectives for group 3 specify something about efficiency for the experienced user.

Tasks

A task is a piece of work that the user wants to perform with support from the system. The task must be *closed*, that is with a limited duration and with a well-defined, meaningful purpose to the user.

Readers knowing about *use cases* might notice that a use case and a task is almost the same. The term *task* is preferred among experts in human-computer interaction while *use case* is preferred among experts in object-oriented analysis.

Which tasks do we choose as basis for the requirements? We recommend that you specify usability for all critical tasks, i.e. those that are closely related to the higher-level goals of the system. This is the reason for the tasks chosen in the ATM case. The justification for each requirement shows the relation to a higher-level goal.

User Groups

Usually a system has several groups of users with different usability requirements. The ATM example looked only at the customers, but the bank staff and the technical service staff are other user groups with different usability requirements. Usability for them must be specified based on the tasks they perform by means of the ATM, e.g. loading cash into the machine, setting up new security codes for the ATM, diagnosing and repairing faults.

1.1. Verification and Tracing

Although the specifications in example 1.1 are quite precise, they give only a rough guideline for how to verify them in the final system. Are we going to observe users of the final system on the street? How many users do we have to observe? Or are we going to make a usability test? How do we select users for the test? How much help do they get?

In practice it is not sufficient to verify the usability at end of development. It is necessary to estimate it several times during development. A test-based approach in an experimental setting can help us. If we specify verification as in example 1.2, we can use the same procedure during development.

This usability test procedure is not completely faithful to the specification. For instance the samples are quite small, so that requiring that 9 out of 10 persons do something is not statistically reliable. The verification of R1.4 (using three days of experience rather than a month) is just a coarse approximation, since we cannot get real experienced users. On the other hand, the procedure is quite easy and cheap to carry out, and it gives acceptable precision in practice (Lauesen, 1997).

1.2. Getting the Data

In order to set up a performance-based usability requirement, you need to collect three kinds of data: A list of the critical tasks, a list of user groups and experiences, and performance objectives for each task/user combination.

Finding the critical tasks is part of the general analysis of the domain. During interviews, group discussions, and observations you try to identify the tasks to be supported by the system. A complete list of tasks is essential to define the required functionality of the system, but it is usually too cumbersome to define and verify usability objectives for all of them.

So you have to identify the critical tasks. One way is to ask these questions:

1. Which tasks are critical to meet the goals of the system?
2. Which tasks take a large part of the user's work day?
3. Which tasks have to be done under stress?
4. Which tasks are difficult to perform?

The usual analysis of user groups, stakeholders, and work domains will give you a good list of user groups. Additional questions are needed to identify what experience the users have and which usability factors are most important.

Example 1.2: Verification of performance-based usability

Requirements R1.1-1.3 and R1.5 are verified with this usability test: 10 test subjects with experience from other banks, and 10 without previous ATM experience are selected through telephone interviews. In the laboratory they try the two basic tasks and they try tasks causing the system to reject the transaction for many different reasons. Task completion times are measured manually. The percentages are interpreted relative to the sample of 10 persons. For instance 9 of the 10 novices must be able to withdraw cash within 4 minutes.

R1.4 is verified in this way: 3 test subjects from each group are allowed to practice withdrawal six times. They are asked to come back three days later and try withdrawal again.

It is important that users get no other help than what they would get in a real-life situation.

Based on the critical tasks, the user/experience groups, and the importance of the usability factors, you can easily write requirements in the style of example 1.1.

But from where do you get the target values, e.g. the time to learn a task? If you set up unrealistic requirements, nobody will enter a contract to provide the system, but what is realistic? If you specify too weak requirements, the users may be annoyed with the system or the customer's business may be slowed down. Here are some ways to get the figures:

1. Are some figures critical to the system goals? For instance a certain total task time might be essential to serve the customers or react to failures. Or you cannot allow more than x days to make the transition from the old system to the new, so the learning time must be less than x days.
2. What is the present performance in these tasks? If you don't want the new system to be slower, you can specify the present performance as a limit.
3. Visit other companies that use similar or related systems and see what performance they get, what learning time they experience, etc.
4. If you plan to buy a standard system, leave the figures out and ask the supplier to provide them. He should have experience that allows him to specify for instance the learning time for his product, various task times, etc. In a tender situation, you may compare the figures from various vendors and use them in the selection procedure.

1.3. Pros & Cons

Usability consists of several factors, ease of learning, efficiency, etc. We can cover most of them with the performance-based approach.

The understandability factor, however, is somewhat outside the scope of the performance-based approach. When measuring user performance, we see only indirectly whether the users understand what is going on. We can, however, ask users specific questions relating to the task and count the correct answers to get a performance measure for understandability. Requirement R1.5 in the ATM case is an example.

The last usability factor, subjective satisfaction, cannot be measured in a performance-based fashion. Satisfaction does not relate to individual tasks but to the whole work situation.

The main problem with the performance style is that the choice of tasks is critical. If an important task has been left out from the specification, users may find that task so difficult to perform that it impairs the value of the system. Careful requirements engineering can guard against that, however.

Another problem is that tests to measure the task performance give rather little feedback to developers. They may observe that users take a long time to learn how to use the system, but what is the underlying problem? Didn't the user see the correct menu point? Or didn't he understand the error message? The defect style avoids this problem.

2. Defect Style

A variant of the performance style is the defect style. It identifies the usability problems in the system and sets limits on the number of problems and their severity. Example 2 shows an example. It is based on a classification of the problems according to severity.

Example 2. Defect style for an ATM

In their first attempts to carry out tasks A and B, users may not encounter more usability problems than these:

R2.1. Task failures: at most 0.2 per user.
R2.2. Efficiency problems: at most 0.2 per user.
R2.3. Inconveniences: no limit.

Task A: Withdraw a preset amount of cash.
Task B: Withdraw a non-preset amount.

Usability Problems and Usability Defects

When a user makes a mistake or finds the system too cumbersome to use, we have a usability problem. There are many kinds of usability problems and some problems are more severe than others. A task failure, for example, is a situation where the user is unable to complete the task or unable to understand why it cannot be completed. Lauesen (1997) gives a taxonomy of usability problems.

A *usability defect* is a design defect which causes usability problems. The same defect can appear as a task failure to some users, as an inconvenience to others, and isn't noticed by the rest. In practice you can report the observed usability problems as a list of defects with indication of the associated problems.

2.1. Verification and Tracing

In order to verify the requirements or check them during development, you have to observe and count usability problems. The best way is to use prototypes and usability tests with a slight modification: ask the users to think aloud. This slows them down, but gives excellent feedback to developers.

2.2. Getting the Data

You can elicit the requirements in much the same way as for the performance style, but it is much more difficult to target values for problem frequencies than for performance times.

2.3. Pros & Cons

The advantage of the approach is that the requirements can be checked easily during system design by means of usability tests. Properly done this gives excellent feedback to developers about the usability problems.

As with the performance style, the choice of tasks and the choice of users are critical, but the same approaches can be used in both cases.

The main problem is that we don't fully cover the usability factors. For instance we assume that if the user doesn't complain about slow performance, then the system is efficient. We also assume that the user is satisfied with the system and would recommend it to others if he hasn't encountered defects. These are dubious assumptions as discussed in section 4.

3. Process Style

It can be quite difficult to set proper limits on the usability, e.g. the number of defects allowed or the maximum time to learn the system. One way to avoid that is to

specify the usability aspects of the design *process* rather than the product. One design aspect is prototyping, and we have often seen requirements saying "development must be based on prototypes". However, a prototype doesn't guarantee usability unless it is usability tested.

Another pitfall is that the prototype is only taken as a guideline in later development, so that the final user interface is designed by the programmers without further usability testing. Experience shows that this introduces many usability defects. Example 3 shows process-based requirements that try to guard against these pitfalls.

You can add additional requirements to ensure that the right users and the right tasks are used in the usability tests. You might also add that after the last usability test, the customer decides whether to use the last version of the prototype or pay for more redesigns and tests.

3.1. Verification and Tracing

Verifying the process-based requirements is a matter of checking that the development process proceeds as specified. It is normally a part of the general quality assurance. A useful approach is to collect the various prototypes to see that appropriate changes have been made and to inspect the problem logs resulting from the usability tests.

3.2. Getting the Data

How do you know that the process-based requirements result in the desired usability? This depends on the actual process, of course. If the process is to inspect the interface or check that standards have been followed, the process cannot guarantee usability. But with the process outlined in example 3, we can get a high level of usability, provided that we have experienced developers and use an adequate number of iterations. Actually, the iterative prototype technique with usability tests is the best known way to ensure usability.

If you have chosen the process, the only other data you need is the proper number of iterations. It is difficult to choose, but three is definitely the lowest number that gives significant effect. Professional interface developers say that they often need six iterations to reach the desired usability level. So it is a good idea to leave an option for the customer to influence the number of iterations.

3.3. Pros & Cons

A major advantage of the process style is that you don't need limits on defects, learning time, etc. You leave that to subjective judgement during design. With the current state-of-the-art, however, there is an even larger advantage: Suppliers are reluctant to guarantee any absolute level of usability, but they might commit to an iterative, prototype-based approach. This means that in many cases the process style is the best you can specify.

The weakness of the approach is that developers need skill and experience to

Example 3. Process style

R3.1: During design, a sequence of 3 prototypes has to be made. Each prototype must be usability tested and the most important defects corrected.

R3.2: During programming and testing, inspection must be made to ensure that the prototype is accurately implemented in the final system.

follow the process and obtain the desired result. Experience also shows that development rarely is truly iterative. The first prototype tends to be the basis for all the later prototypes, which become minor modifications of the first. In contrast, the best designers often start from scratch, rethinking the entire approach after the first usability test (Lauesen, 1997).

4. Subjective Style

In the subjective style we define a set of criteria for satisfaction with the system. Example 4 shows a usability requirement in this style for an ATM. It catches quite well the usability factor called *subjective satisfaction.* You can define criteria for other usability factors too, such as ease of learning, task efficiency, etc. in a similar manner by asking users whether they find the system easy to learn, efficient for their daily tasks, etc. In principle, the style can come very close to the definition of usability.

4.1. Verification and Tracing

When the system has been in operation for some time, you can verify the subjective requirements. You have to design a questionnaire and ask a suitable sample of users to complete it. To verify R4.1 you could simply ask the users to what extent they find the system pleasant to use and whether they would recommend it to their friends.

4.2. Getting the Data

In order to set up subjective-style criteria, you have to find the right subjective factors and the right percentages of satisfied users. Little is known about how to do that, but we would expect that the subjective factors should be closely related to the overall goals of the system. In the ATM example, for instance, a major goal for the bank was to promote itself through the ATMs. The subjective criteria were selected to support that.

Finding the right percentages is more difficult. What are realistic objectives? Probably the best approach is to study satisfaction with similar products, decide whether that is sufficient, whether it seems possible to raise the level of satisfaction, and then select the necessary level.

Some suppliers investigate satisfaction with their products, and you could use their figures as a basis. In a tender situation, you might ask the supplier to provide the figures and use them as part of the total evaluation of the proposals.

4.3. Pros & Cons

Compared with the performance and defect styles, the subjective style is not dependent on defining the correct tasks. It covers whatever work situation the users are in. Further, it is the only style that directly can cover the subjective satisfaction factor.

A major problem is that it is difficult to check the requirements during devel-

Example 4. Subjective style

Customers associate us with the ATM and it is important that our image is supported by the system.

R4.1	80% of customers having tried the ATM at least once must find the system pleasant and helpful. 60% must recommend it to friends if asked.

opment. Making prototypes and usability tests can give us early information about task times and usability problems, but nobody knows at present how that relates to the subjective satisfaction. The best approach is probably to end each usability test session by asking the user to complete a questionnaire like the one above. Many usability specialists practice that, but it is not known how well it predicts satisfaction in real use with all the task variations that were not tested in the usability lab.

Another problem is that a lack of subjective satisfaction is difficult to deal with. If users are dissatisfied, developers don't know what changes to make. What are the causes of the dissatisfaction?

Subjective satisfaction is not solely a matter of the right user interface. The way the system is introduced, the general motivation and stress level of users, and other organisational factors seem to have a dominating influence. For instance, we have observed that users can state high satisfaction although observations of their interaction with the system show that the system is awfully slow and often cause problems that the user doesn't know how to deal with. From a management point of view, the system wastes human labour, but users seem satisfied.

This puzzling fact might have several explanations. In some cultures, people don't criticise, and they state satisfaction even if dissatisfied. In cultures where people readily criticise, they may be proud of mastering the system in spite of its shortcomings, and the pride results in a positive evaluation. Nielsen & Levy (1994) have compared several studies of subjective satisfaction against performance factors and found weak correlations.

5. Design Style

The traditional requirements style is to specify the screen pictures and screen functions, for instance as a prototype. Example 5 shows requirements based on a finished design in the form of a prototype.

Essentially, this approach has turned the usability requirements into functional requirements. The requirements engineer has taken responsibility for the ease-of-use, and the designer and programmer can do little to change it. In some situations this is the right approach; but if we also specify usability, e.g. time to learn or task efficiency, we will most likely have a contradiction in the requirements.

5.1. Verification and Tracing

Design-based requirements are easy to verify in the product and easy to trace during development. For instance, you can inspect the final interface and the design artifacts to see that they accurately reflect the prototype. Inspection is quite important because developers often consider the prototype a guideline, and they design something different without further usability testing. Experience shows that this introduces many usability defects.

5.2. Getting the Data

If you use a design style, how do you get the data, that is the design? There is only one feasible way: You have to use prototyping and usability tests, but as part of the requirements engineering.

Example 5. Design style

R5.1: The system shall use the screen pictures shown in App. xx.

R5.2: The menu points and push buttons shall function as shown in App. yy.

5.3. Pros & Cons

The design style eliminates the need to specify usability factors such as ease of learning or task performance. The price is that the requirements team has to ensure the necessary degree of usability in the prototypes.

What we often see are untested prototypes as part of the requirements. Apparently the requirements engineers thought that they could inspect the prototype to check that it had the necessary usability. Nobody can do that at present, not even usability experts.

Another common problem is prototypes made by participative development where users became absorbed by their own creations. They forgot the real goals and the tasks to be supported, and the resulting system had serious functional as well as usability defects. The remedy is to state the usability goals and essential tasks early on, and frequently verify that the system meets the goals and supports the tasks.

6. Guideline Style

It is a common belief that if you follow user interface style guides and standards, you get high usability. Example 6 shows usability requirements based on this assumption. Although guidelines are quite useful, they are very far from ensuring usability.

6.1. Verification and Tracing

It is possible but not easy to verify guidelines in the product and during development. Some guidelines are supported by development tools. In other cases you must inspect the final interface and the design artifacts to see that they accurately reflect the guidelines.

Inspection is not that easy because there are many rules in the guidelines (typically several hundred), or the guidelines are quite broad so that they have to be interpreted in each case. Unless inspectors are well trained, many defects are missed in such inspections, or they give rise to debate about whether a specific rule is violated or not.

6.2. Getting the Data

If you use a guideline style, how do you get the data, that is the guidelines? There is no simple answer to this. One important factor is the user's background. Are they accustomed to applications in a specific style? Is it important to reduce switching costs? Is it likely that other products following a specific style will be introduced. Answers to these questions may help decide the required standard.

Available standards such as MS-Windows or CUA are helpful, but more specific rules are often used. Some companies maintain a list of additional rules triggered by usability problems in previous products. Requirements R6.2 and R6.3 are examples of such experience-based rules. After some time, however, the list tends to be too long

Example 6. Guideline style

R6.1: The system shall follow the MS-Windows style guide.

R6.2: For input fields with a limited set of values, it must be possible for the user to select the value from a list.

R6.3. All dialogue boxes must be non-modal so that users can look at other windows while responding to the dialogue box.

R6.4. The interface must resemble the interface of application xx.

for practical use.

Other companies develop a domain-specific standard, for instance for all their business applications. A good approach is to develop a prototype of such an application, usability test it, and then use the prototype as a guideline. This works well in many cases, and an additional advantage is that developers find it more easy to use an example than a set of rules.

6.3. Pros & Cons

Guidelines can be great, particularly to help users switch between many applications. However, in general, guidelines have little relation to how easy the system is to use. In other words, you can have a system that users find very hard to use although it follows the guidelines. Such systems are actually quite common, as demonstrated by the many programs that follow the MS-Windows guidelines, yet are very difficult to use.

Experiments have shown that checking a design against a good guideline can reveal about 25% of the real usability defects. The checking process also finds a lot of guideline violations that are not really problems to the users (Cuomo & Bowen, 1994; Desurvire et al., 1992; Jeffries et al.,1991).

7. Match with Requirement Scenarios

In practice we meet different requirement scenarios, e.g. requirements for product development versus requirements for a tender process. Below we will summarise the usability styles useful in common requirement scenarios.

7.1. Product Development

1. The performance style is useful, and it doesn't have to be very exact since developers and marketing can modify and interpret the requirements along the way.
2. The defect style is more useful since it provides better feedback to developers.
3. The process style (with iterative prototypes and usability tests) is equally useful. The number of iterations can be adjusted during development according to the outcomes.
4. The subjective style is less useful, but it can be used for setting overall goals. You can deviate from them if it is too difficult to fulfill them. The criteria can be quite useful for marketing, and planning the next release can get important input from measuring the criteria in the previous release.
5. The design style is useful, but in order for it to work, you have to do a lot of design work and usability testing during requirements specification.
6. The guideline style is useful as a supplement to other styles.

7.2. In-house Development

The styles can be used in a similar way as in the product scenario. Essentially, the user organization serves the role of marketing.

7.3. Contract Development

1. The performance style is useful, and in some cases you have to be very exact about how to verify the requirements. Many vendors will not commit to it, however.
2. The defect style is useful too and it provides better feedback to developers. Still, many vendors will not commit to it.

3. The process style (with iterative prototypes and usability tests) is very useful and most vendors can commit to it.
4. The subjective style is hardly useful. Few vendors would be willing to provide usability according to the subjective criteria. This is not only a matter of state-of-the-art in usability, but also the fact that the vendor has no influence on the organizational factors at the customer site.
5. The design style is possible, but in order for it to work, you have to do a lot of design work and usability testing during requirements specification. The approach is particularly useful in cases where a main contractor designs the system, and the sub-contractors get the usability specifications in the form of prototypes.
6. The guideline style is useful as a supplement to other styles.

7.4. Tender with Standard System

1. The performance style is useful, and it doesn't have to be very exact since a more detailed specification might be made as part of the contract. It may be difficult to select the actual performance figures, but ask the vendor for the figures and compare the figures from different vendors.
2. The defect style is useful too, but it is more difficult to get reliable figures from the vendor due to different interpretations of defect severity.
3. The process style (with iterative prototypes and usability tests) is not useful since the standard system has been developed already. However, for additions to the standard system, you may use the process style.
4. The design style is not useful for the same reason.
5. Few vendors would be willing to provide usability according to the subjective style, but it is possible to ask them for figures about the actual subjective satisfaction with their product. The customer can then compares the vendor's figures.
6. The guideline-based style is not useful since the standard system most likely has its own style.

References

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.

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.

Lauesen, S. (1997): Usability engineering in industrial practice. In Howard et al. (eds.): Human-Computer Interaction, Interact'97, Chapman & Hall, pp. 15-22.

Macaulay, L. (1996): Requirements engineering. Springer.

Nielsen, J. (1993): Usability engineering. Academic Press.

Nielsen, J. & Levy, J. (1994): Measuring usability, Preference vs. performance. Communications of the ACM, 37(4), pp.66-75.

Preece, J. (1994): Human-computer interaction, Addison Wesley.