

A photograph of a man and a woman in business attire looking at a computer monitor. The woman is on the left, holding a rolled-up document. The man is on the right, wearing glasses. The background is slightly blurred.

Intelligent tutoring systems for computer software

Or... teaching software to understand humans...?

Bart Knijnenburg, Carnegie Mellon University

Abstract

In this paper I propose a new paradigm for human-computer interaction: The use of intelligent tutoring systems to negotiate the interaction between a user and a user interface. I describe how such an approach would work, and how we can make usable interfaces using this approach.

The paper first highlights the common practices and techniques for creating usable interfaces. I will then show why even appropriately designed systems are

These programs are only intelligent because they make us users feel incredibly stupid.

not always usable. The paper will then briefly touch upon teaching computer software. Intelligent tutoring systems are then introduced, and analogies are drawn between intelligent tutors and both real life tutors and a structural understanding of the interface design problem.

The paper will then describe how an intelligent tutoring system for computer software would work, what the benefits are, and what might possibly impede their success. Finally, I will explain the mystic subtitle of this paper.

Introducing the problem

How to make usable interfaces

In today's urban jungle we are constantly being attacked by vicious technology and malicious user interfaces. In our fast-paced lives, there is often no time to learn how to deal with these artifacts of modern society. We all need to instantly know how to deal with yet another horrendously designed ATM, elevator panel, or ticket machine. Even the devices we own and interact with over a prolonged period of time seem to mystify us beyond frustration: try programming your VCR, your thermostat or the clock on your microwave oven.

Even in the more defined space of computer software, we have a hard time learning new programs, and I think you know the nagging suspicion that you're doing things inefficiently or in a completely unintended way whenever you use a Microsoft Office product. Some companies are starting to call their software "intelligent", but in my view these programs are only intelligent because they make us users feel incredibly stupid.

Current solutions

Usability evaluation methods

Fortunately, there is a group of heroes to the rescue. Combining computer science, software engineering and cognitive psychology, usability engineers try to improve user interfaces. They use a variety of methods and techniques – founded in cognitive science research – to improve our interaction with the systems in our lives.

Usability design cycle

Usability engineers know that dealing with humans requires a flexible approach. Instead of designing software in one go, they use an iterative approach of several “design and test” cycles. Since these cycles take about a week or so, the tests are not as meticulous as empirical tests in psychology. A number of “discount” usability methods have been devised in order to fit the testing into such short design cycles. [Nielsen, 1993, chapter 4]

Usability evaluation methods

The usability evaluation methods that are invented all have to meet (and have been thoroughly tested to meet) a set of important criteria. First of all, since not all methods require actual test-users interacting with the system, the methods should reflect the results of real usage. Second, since the methods are reasoning about human behavior, the theory behind the methods should be founded in cognitive science research. Third, the methods themselves need to be usable, since they need to be applied correctly and extensively within the short design cycles.

In **think aloud** [John, 1999] method, an experimenter asks users to think aloud while they are performing a task on the interface that is being tested. The users are asked to verbalize their imperceptible behavior, like “I want to set the timer, so I’m looking for a button that says timer or something” or “I’m looking for the record button”. This way the experimenter can

follow all the steps the user takes in working with the interface. The users are not allowed to explain their behavior. Explaining behavior is called introspection, and cognitive science research has found that people’s introspective reports often don’t match with their real cognitive processes [source]. Besides that, it has been shown that the very act of introspection changes people’s behavior significantly. So using introspection would cause our results to mismatch with real usage.

Fortunately, there is a group of heroes to the rescue.

Jacob Nielsen devised the **heuristic evaluation** [Nielsen, 1993, chapter 5, Nielsen, 1994] method by combining and condensing the results of a large number of empirical user tests. The idea behind this approach is that real user problems are caused by underlying cognitive phenomena, and that knowledge of these phenomena can help you design better interfaces. Nielsen devised 10 heuristics that can help you prevent or detect and solve usability problems. In order to keep them simple, these heuristics don’t actually point out the underlying cognitive phenomena. Having some knowledge of cognitive science helps a lot in using heuristic evaluation, because the heuristics “feel” more natural when you know about cognitive phenomena.

In the **cognitive walkthrough** [Wharton et al, 1994] a group of usability engineers walks through a task on a user interface, reasoning about whether a first-time user of that interface would be able to do this task. For each step that this imaginative first-time user has to take, the engineers ask themselves four questions:

- Will the user try to achieve the right effect?
- Will the user notice that the action is available?
- Will the user associate the action with the effect?

- Will the user see progression? [Wharton et al, 1994, p112]

Since the user is not familiar with this interface, but most likely has knowledge about how to use other interfaces, these questions are mostly about whether the user can transfer the knowledge of these interfaces to this new interface. This transfer is most likely done by structural analogy, a process in which

Making usable systems seems to be a next-to-impossible task.

the user matches the structure of a new problem to an old (or generalized) problem and substitutes the specifics to match the new problem. The four questions address how such analogies show up in goal structure, perception, cognition, and feedback respectively.

The **keystroke-level model** is designed to optimize user interfaces for expert users. Card, Moran and Newell [1983] defined a task as being composed of small parts (in their smallest version, KLM-GOMS, these parts are keystrokes, button-presses, mouse-clicks, etc.). They measured the best possible performance of these small parts, and then reasoned that the performance of the whole task could be measured by adding the performance times of the composing parts together. This method makes the assumption that tasks can be broken down into small parts, and that the performance is additive. The keystroke-level model actually underpredicts the performance of real experts. It is well-known in cognitive psychology that some basic acts can be performed in parallel. John and

Atwood [John, 2003] made a better model, CPM-GOMS which takes into account the possibility of parallel behavior.

Even when the aforementioned evaluation methods are properly used, making usable systems seems to be a next-to-impossible task. In order to see why this is the case, we have to impose a conceptual structure on the “usable systems problem”. This will be done in the next section.

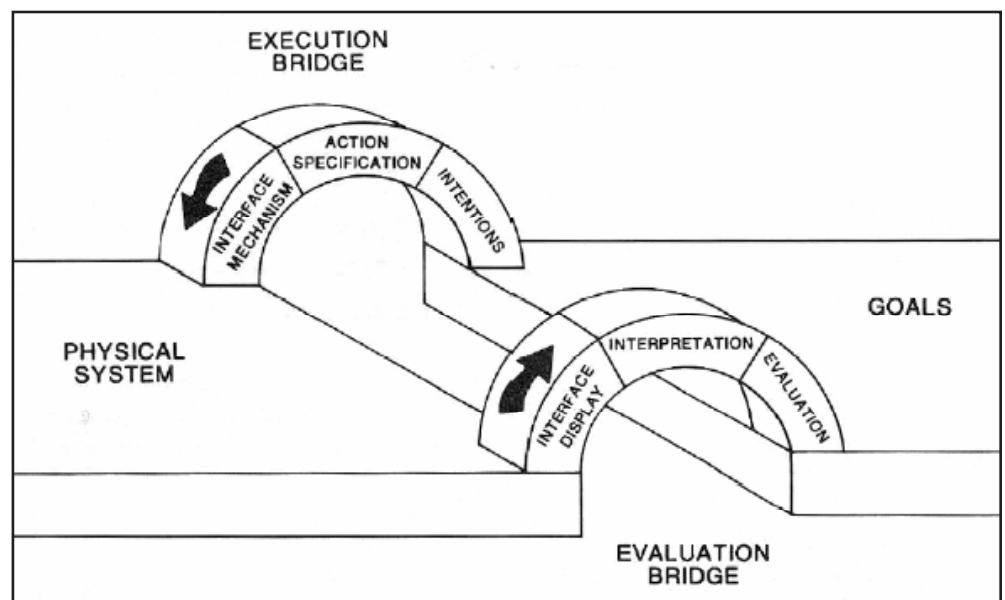
The problem revisited

Norman’s 3-model approach

The Gulf of evaluation and execution

At the most basic level possible, there is one big problem that these usability evaluation methods all try to solve; this is the problem of a user not being able to do a certain task on a certain interface. Taking this bottom-up approach, Donald Norman [1988, pp45-53] argued that there are two “gulfs” between the user and the system.

The first gulf is the **gulf of execution**. The user starts out with a certain goal that he/she wants to accomplish using the interface. From this goal, the user has



to form an intention to act upon the interface, and this intention must be translated into the specification of an action. The user then performs this action on the interface, and the system starts working. In this process, anything can go wrong: not knowing what next step to take, not knowing how the next step translates into an action, or not being able to perform the action on the interface.

Designing a good user interface, therefore, is making the gulfs as small as possible.

The second gulf, the **gulf of evaluation**, occurs after the system did something in response to the user action. In order to find out what the system exactly did, the user has to notice some change in the state of the interface. The user then has to interpret the meaning of that change, and evaluate whether this was the effect intended. This evaluation ties back to the goal, because the user now knows whether he/she is closer to fulfilling this goal. In this part of the interaction there are also a lot of possible breakdowns: not seeing a change in the interface, not knowing what this change means, or not being able to relate the meaning of the change to the goal.

In using a system, users are constantly trying to bridge the gulfs of execution and evaluation in order to fulfill their goals. Designing a good user interface, therefore, is making the gulfs as small as possible.

Designer model, use model, and system image

Of course, we would not be human beings if we would not have a way to conceptualize this problem in our minds. In fact, this conceptualization occurs in both the designers of the system and the users of the system [Norman, 1988, pp12-17].

The designers of a system work from requirements specification to design a system that fulfills the re-

quirements. They make an alarm clock that shows the time, has an alarm, a radio option, a display dimmer, and a way to set the clock, the alarm and the radio frequency. They draw (in their heads or explicitly on paper) a **designer model** of the system. This model maps the functionality of the system to the interface of the system. Now, when the system gets built it has a **system image**, which is an implicit model of its behavior that matches the designer model. The users of the system conceptualize by creating a **use model**: from the appearance of the interface and their reflection on their interaction with the system, they derive their own model of how the system works.

Ideally, the use model is exactly the same as the designer model and therefore the system image. When the user succeeds to perfectly match its use model to the system image, the gulfs of execution and evaluation are virtually non-existent: the user knows exactly how to control the system and interpret its outcomes. The problems start when the use model is not aligned to the system image: that is exactly the point where the interaction breaks down, because the user cannot bridge the gulfs. Designing a good user interface, therefore, is making the gulfs as small as possible, by making a system image that can be easily interpreted and translated to a correct use model.

When the user matches its use model to the system image, the gulfs of execution and evaluation are virtually non-existent.

Affordances and natural mappings and why they fail

Norman gives us a two-part solution to designing good interfaces. On the interactor level (the level of single buttons, scrollbars, menus), he advocates good **affordances** [Norman, 1988, chapter 4]. Affordances are inherit properties of interactors that

entice a certain way of interacting with them, and that give proper feedback of that interaction. On the system level, he suggests designers to use **natural mappings** [Norman, 1988, 75-79]. Natural mappings are interface layouts that are map to the natural state of things. For instance, have the “volume up” button above the “volume down” button. A good example of a breakdown because of the lack of natural mappings is a regular stove top: the burners are laid out in a square, while the knobs to control the gas flow are in one line. In this case it is hard to find out which button is for which stove, since the layout of the buttons does not naturally map to the layout of the burners.

Alas, both affordances and natural mappings tend to break down in modern user interfaces. As Dja-jadiningrat et al. [2002, p1] point out affordances don't work for the multi-faceted interaction we have with modern electronic devices. Although buttons and sliders have their affordance, they only help in the action specification part of the gulf of execution. With complex systems for complex tasks, however, it is the formation of an intention that often fails (“what should I do next”) and affordances do not help here. In the same way natural mappings break down: there is nothing natural to most of the tasks we perform and simple reasoning by natural analogy often breaks down when the task gets more complex.

Both affordances and natural mappings tend to break down in modern user interfaces.

The problem of human variability

Another problem that usability engineers encounter has to do with the variability of human cognition. For a system with anything but the least complexity, it is highly unlikely that all users have the same (correct) use model. There is often a variety of ways to interpret a systems interface and the outcomes of interaction,

so it is quite possible that these interpretations lead to a wide variety of use models. Therefore, it is simply impossible to design a system that everyone understands: Make a system model that works for person A, and person B will just not get it.

Real life solution

Tutoring computer software

So it seems that even in a world full of usability engineers and well-designed systems some people will still have trouble using these systems, because they don't understand the system image. What is the best thing to do when you want to learn something you don't understand? – You take a class in it!

For many years I have been a teacher of various computer courses. I taught from basic Windows to advanced Excel. I taught to high school kids and grandfathers. I basically used two approaches to teaching, which I will describe below.

Group teaching: learning from books

The “cheapest” approach is the group teaching approach. Students come to class every week and learn to master a certain software program in about ten weeks. They have a book with instructions and assignments. The lectures are more like work sessions, where I briefly clarify a new topic, and then help students with their specific problems. As expected, the problems are extremely varied: what is almost insultingly easy for one person can be almost ungraspable for the other. When I probe on problematic situations, virtually all of the problems are due to inadequate use models. Fortunately, as expected, point out the explicit use model work extremely well as a way to teach.

The main problem with the group teaching approach lies in the book. It is almost impossible to teach someone a system model from scratch. For

complex systems like any Microsoft Office product, it is impossible to point out every single part of the system image. Therefore, most books make an assumption about the existing use model of the students, and work from there to augment the use model with parts of the system image. Students waste valuable time reading about a part of the system image that they turn out to already know, and they have problems when the book rushes over a part of the system image that they find difficult to understand in one go.

One on one tutoring: a goal directed tutoring

A solution to the aforementioned problems is to do one on one tutoring. This teaching style has many advantages. By interpreting the students' actions with the interface, I can find out exactly what parts of their use model are incorrect. This allows me to give students directed feedback on their inaccurate use models, providing them with their knowledge that corrects these mistakes. Besides that, I can monitor the students' learning progress and give them appropriate assignments that fit their knowledge level. This speeds up the learning incredibly. Also, the student can ask me to teach specific parts of the interest that are of the most interest to the student.

One on one tutoring works a lot better than group teaching, but it still has two problems. One problem is that this method is extremely expensive, since the cost of the expert is now on one person instead of an entire class (about 12-18 students). The other problem is that the tutored students can only get help on appointment; they cannot easily access the tutor whenever they have a problem during their work.

Two analogies

Intelligent tutoring systems

What is an intelligent tutoring system?

Intelligent tutoring systems (ITSs) were invented by John Anderson around 1983 [Anderson et al, 1995, p171]. Over the past decades, these systems have been shown to successfully teach programming, geometry and algebra. ITSs use a "teaching by doing" approach, the ITS selectively presents problems to students, and corrects the students if they make mistakes. In order to do this, intelligent tutoring system has three main important parts:

- An expert model
- Model tracing
- Knowledge tracing

The expert model is the model that the system has of the solution to the problem at hand. This solution is often implemented as a set of production rules in the ACT-R environment and mimics the steps a knowledgeable person would take to solve the task [Anderson et al, 1995, p171, p179, Anderson and Gluck, 2000, pp4-7].

Model tracing means matching the observed behavior of the student to productions in the expert model [Anderson et al, 1995, p172, Anderson and Gluck, 2000, p7]. This way the tutor can understand why the student did something wrong, and show the student the correct step, or give the student a hint.

Knowledge tracing means figuring out the competence level of the student. The Lisp tutor [described in Anderson et al, 1995, p187, Anderson and Gluck, 2000, pp7-8] used a Bayesian procedure to calculate the probability of a student knowing a certain production rule. Using knowledge tracing, the tutor can gradually introduce new concepts and strengthen old ones.

Analogy one: automated vs. real life tutoring

It is clear that ITSs match the approach of real tutors. First of all, they both increase learning. In a high-school or college environment, real tutors have been proven to increase test performance by about two standard deviations and the tutors described in Anderson et al [1995 p177, p183] and Anderson and Gluck [2000, p9-11], seem to improve performance by at least one standard deviation. Besides that, drawing from my work as a computer tutor, it seems that model tracing (figuring out why a student makes a certain mistake) and knowledge tracing (figuring out what to present to the student in order to advance him) are exactly the strategies that I use to tutor computer software. Concluding, the ITS approach seems very appropriate for tutoring computer software.

Analogy two: models

There is another, deeper analogy present here. The expert model of a tutoring system for software would essentially be the system image (or the designer model for that matter). Model tracing is in this case actually interpreting the gulfs that exist between the user and the system, and the help that a tutor offers basically shows the user how to bridge the gulfs. Concluding, intelligent tutoring systems can solve the usable systems problem as defined by Norman.

Evaluation

ITSs for computer software

Solving the problems

The two analogies give us good hope that intelligent tutoring systems can solve the usable systems problem. In this section I will explain how I envision such a tutoring system to work.

An ITS for computer software should first try to figure out what the goal of the user is. Depending

on the amount of freedom allowed to the user, this will not be a trivial problem. In the most restrictive scenario, the user has to choose a task from a set of predefined tasks at the beginning of the task. This scenario would fail for most complex software systems, since there are almost infinitely many tasks to perform on the system. Another approach is to let the user specify the task in his/her own words. In this case the ITS must interpret the task description and derive a goal from that interpretation. In the ideal case, the user would just work on the system, and the ITS would constantly try to figure out what the user's goal is based on the recently performed actions. This may very well be extremely difficult, and any failure or mistake in deriving the user's goal will most likely deteriorate the tutor's performance.

ITSs can solve the usable systems problem as defined by Norman.

When the ITS figured out what the user's goal is, it must match the user's actions to its own production rules for that goal. This is basically the same thing as model tracing. Most likely, there are many different ways of performing the task on the system, and the ITS must of course account for all of them. As in other ITSs, our ITS must also have production rules for incorrect actions. These false production rules then link back to a misconception in the underlying use model.

When the user takes an incorrect action, the ITS can either correct the mistake automatically, or derive the misconception in the use model and correct this misconception by explicitly teaching the user about the system image and how he/she got it wrong. The first option has the advantage of being unobtrusive, but the user will not learn from his mistake, and might probably even be confused by the correction. The second option is more intrusive, but in this case

the user will actually learn from his/her mistake, and his increased understanding of the use model can help him/her in other tasks too.

Since the ITS is not presenting the user with assignments, knowledge tracing seems to have no purpose in this application. However, the tracing of the use model seems to lend itself perfectly for a Bayesian approach, where the probability of several false and correct assumptions about the system are calculated based on the user's actions. It is not only important to find out whether or not a production in the use model is correct; also of importance is the fact if a certain production is actually present. Most users actually start out with an oversimplified version of the use model, and gradually expand the model (in a correct or sometimes incorrect direction) when they go along tackling more difficult tasks. The tutor could calculate the optimal amount of help to offer in a certain task based on the difference between the expertise of the user and the difficulty of that task.

Other advantages

In the last section I described how an ITS could help people using software. The ITS is a cheaper alternative than a real-life tutor, and drawing from previous successes of ITSs, we can predict favorable results of such an approach. Besides that, there are several reasons why an ITS solution to the usable systems problem may be especially favorable.

Solving a problem involves decomposing the problem into a set of goals and sub-goals.

First of all, an ITS provides on the spot instructions. Whenever the user needs help, he/she can ask the tutor for help. The tutor can also suggest a correction whenever it is fairly confident that the user made a mistake [Anderson et al, 1995, pp180-181]. The user's use model is being tuned while the user is doing his

work, so the training does not cost any extra time or effort.

As said before, the gulf of execution often exists because the user is unable to form the right intention. The user just doesn't know what to do next. In the high-school usage of ITSs, this problem is called task decomposition: solving a problem involves decomposing that problem into a set of goals and sub-goals. In many domains, students have problems when the

The ITS approach takes into account the variability of the user.

goal structure is not adequately communicated to the student. ITSs solve this problem by exposing and communicating the goal structure [Anderson et al, 1995, p179]. The same thing can be done with ITSs for computer software; whenever the ITS figures out what the high-level goal of the user is, it can propose a goal structure that helps the user define the appropriate intentions. I call these goal structures meta-affordances, since in a sense, they provide an overview of what sub-components the task involves.

Last but not least, the ITS approach takes into account the variability of the user. By default, it assumes that every user has a different, not necessarily complete or correct use model. It solves the user's problems with the system by finding the incorrectness and incompleteness of this model and correcting it, bringing it closer towards the system image. For an analogy with conventional ITSs, see Anderson and Gluck [2000, p36].

Possible points of failure

When discussing the possible points of failure of an unexplored technology, the first reasonable thing to ask is: "Why has no-one ever tried this before?" Actually, it has been done before, by Microsoft, and it is called Clippy. Clippy is probably the most hated

interface invention of all time. Clippy was such an annoying thing, that it likely caused other companies to shy from using agent-mediated interaction. A big problem of Clippy, however, is that it assumes you're equally ignorant or knowledgeable across all of Office's applications [Spark, 2001]. It is basically a very dumb tutor in that respect. What we can learn from this, is that just because an idea is implemented in a bad way, it does not immediately follow that the idea is bad and should not be pursued anymore.

"Why has no-one ever tried this before?"

However, there are some appropriate reservations to be made when considering ITSs for computer software. First of all, people using software are not students. They are not just learning how to use software; they are also fulfilling their task. If the intelligent tutor is too intrusive, this will interfere too much with the task at hand. The main intrusion of a tutor is the feedback it gives. Unfortunately, it has been shown that more intrusive feedback provides better learning results; we seem to have a trade-off here [Anderson et al, 1995, pp188-191]. Will the ITS stay on the background, ready to help at the press of a button, or will it intervene when the user strays off the path? In case of the latter, the ITS has to carefully decide when to intervene and in what way. In some cases quietly flagging the problem will work, in other cases a more drastic intervention is needed to get the user back on the path. If the error is explained to the user, the error message needs to be tuned to the user's current use model.

As said before, finding out what the user's goal is, is a very difficult but important task. It is only when the ITS knows the user's goal that it can start the model tracing procedure that makes it an intelligent tutor. From the tutor's perspective, it would be most convenient if the system has a defined (but not neces-

sarily finite) set of tasks that can be performed. This might be taken a bit further to a scheme which I will call "competence-based software": instead of providing the user with just a tool that can be used for whatever the user wishes to do with it, a software company could define its systems as "bundles of competences", in which each software package is defined by the set of tasks (competences) that can be performed on the interface. Most software programs nowadays allow such competence-based usage by having templates and wizards. The problem here is that these templates and wizards take away much of the creativity inherent in most software packages. It is exactly the fact that the software can be used in creative and unintended ways which makes the software so powerful. It seems to be a bad idea to give this up, but it almost seems like such generative usage is inherently contradictory to the ITS approach.

Wild idea

Reversed tutoring

All in all, it seems that intelligent tutoring systems for computer software are a promising area of research. Since this conclusion is not worth an entire section, I will devote the remainder of this section to clarifying the somewhat mystic subtitle of this paper.

The fact that software can be used in creative and unintended ways makes it so powerful.

Working on the proposal for this paper, I was rather thrilled about the idea of having an intelligent tutor mediating the communication between the user and the system. But then I started thinking: "Why do I want to adjust the use model to match the system model? Why not do it the other way around?" I realized that I had thoughtlessly mapped the system

image to the expert model, making model tracing a case of altering the use model. I also realized that it would be radically different if I would map the use model to the expert model, and have model tracing adjust the system image. This would mean that the ITS still tries to interpret the user's use model, but then instead of altering this model to match the sys-

I will devote the remainder of this section to clarifying the mystic subtitle of this paper.

tem image, it would alter the system image to match the use model: adapting the software to the user!

This approach – which I call “reversed tutoring” – may very well be much more powerful than the ITS approach proposed above. Changing the system is definitely a lot less intrusive than changing the user. Also, in the case where we want to change the system instead of the user, it is not insurmountable to have a somewhat shaky understanding of the use model: it is already a great improvement if the system model “somewhat” matches the use model. Furthermore, changing the system might not be as hard as changing a user. People are generally resistant to change, and from the user's perspective it seems quite reasonable to ask the system to adjust to the user instead of the other way around.

Of course, reversed tutoring is not the holy grail. For one thing, use models often start out being rather incoherent, but it would be a fallacy to derive from this that we should make the system incoherent too. In reality, normal and reversed tutoring would work together to optimize the user experience and solve the usable systems problem.

References

Anderson, J. R., Corbett, A. T., Koedinger, K., & Pelletier, R. (1995): Cognitive tutors: Lessons learned. In *The Journal of Learning Sciences*, 4, 167-207.

Anderson, J. R. and Gluck, K. (2001): What role do cognitive architectures play in intelligent tutoring systems? In Klahr, D. and Carver, S. M. (Eds.), *Cognition & Instruction: Twenty-five years of progress*, pp. 227-262. Erlbaum.

Card, S.K., Moran, T.P. and Newell, A., (1983): *The Psychology of Human-Computer Interaction*, Erlbaum.

John, B. E. (1999): Carnegie Technology Education Course #SSD4 User-Centered Design and Testing, Unit 3 Think-aloud Usability Testing.

John, B. E. (2003): Information processing and skilled behavior. In Carroll, J. M. (Ed.), *Toward a multidisciplinary science of human interaction*. Morgan Kaufman.

Nielsen, J (1993): *Usability Engineering*. Morgan Kaufman.

Nielsen, J (1994): Heuristic Evaluation. In Nielsen and Mack (Eds.), *Usability Inspection Methods*. John Wiley & Sons, Inc.

Norman, D.A. (1988): *The Design of Everyday Things*. Basic Books.

Spark, D. (2001): Show The Clip - Reports Of Clippy's Demise Have Been Greatly Exaggerated. In *Smart Computing*, October 2001, Vol.7 Issue 10.

Wharton, C., Rieman, J., Lewis, C. and Polson, P. (1994): The Cognitive Walkthrough Method: A Practitioner's Guide. In Nielsen and Mack (Eds.), *Usability Inspection Methods*. John Wiley & Sons, Inc.