Learning to Use Complex Computer Technology: The Importance of User Interface Design

PhD Depth Paper

by

Joanna McGrenere

Department of Computer Science
The University of Toronto

June 25th, 1998

Supervised by:

Professor Ronald M. Baecker
Department of Computer Science
University of Toronto

Professor Kellogg S. Booth
Department of Computer Science
University of British Columbia

Abstract

Computer systems (and many other devices that contain embedded computers) are becoming more and more complex as advances in technology permit substantially more functionality to be provided to users. Often the vast number of options, and the sometimes ad hoc manner in which these options are invoked, leads only to confusion on the part of the user rather than satisfaction with the product. This can potentially inhibit the user's growth of expertise. A key factor in designing complex systems is the design of an appropriate user interface that exposes functionality in a way that supports the needs of the user as a learner and does not lead to confusion and frustration.

This paper is a literature review that covers two somewhat symmetric approaches to user interface design. The first involves making design decisions based on an understanding of how users learn new complex systems. The second involves having the system learn about the user and then adjust itself or allow adjustments based on what it has learned. This paper surveys previous and current work in both of these areas and draws conclusions about appropriate areas for further research in user interface design techniques based on these two approaches.

TABLE OF CONTENTS

1 INTRODUCTION ... 1

2 SETTING THE STAGE ... 2

2.1 TERMINOLOGY ... 2
2.2 HOW ARE USERS COPING? ... 3
2.3 WHY FEATURISM? WHY THE COMPLEXITY? ... 6
2.4 THE GOAL OF UNIVERSAL ACCESS ... 7
2.5 RESEARCH DIRECTION ... 8

3 LEARNING TO USE COMPLEX SYSTEMS ... 8

3.1 WHAT IS LEARNING? ... 10
3.2 EXTERNAL FACTORS AFFECTING LEARNING ... 11
3.2.1 Motivation ... 11
3.2.2 Individual Characteristics ... 11
3.3 HOW DOES ONE LEARN? ... 12
3.3.1 Exploratory Learning ... 12
3.3.2 Learning Through Transfer of Prior Knowledge ... 16
3.3.3 Formal Training ... 17
3.3.4 User Support - Online Help, Tutorials, Demonstrations and Animations, and Documentation 18
3.3.5 Learning from Friends and Colleagues ... 21
3.4 MODELLING HOW USERS LEARN ... 22
3.5 DESIGNING FOR LEARNABILITY ... 23
3.5.1 Designing for Exploration ... 23
3.5.2 Designing for Error ... 25
3.5.3 Learner-Centered Design ... 26
3.5.4 Questioning the All-in-One Model ... 28

4 INTELLIGENT USER INTERFACES ... 30

4.1 AGENTS ... 32
4.2 ADAPTIVE USER INTERFACES ... 35
4.2.1 Examples of Adaptive User Interfaces: ... 38
4.3 USER MODELLING ... 41
4.3.1 Examples of User Models: ... 43
4.4 TASK MODELLING AND PLAN RECOGNITION ... 43
4.4.1 Examples of Task Modelling and Plan Recognition: ... 44
4.5 MULTIMODAL COMMUNICATION/NATURAL-LANGUAGE DIALOG ... 45
4.5.1 Examples of Multimodal Communication/Natural-Language Dialog: ... 45
4.6 INTELLIGENT HELP ... 45
4.6.1 Examples of Intelligent Help: ... 45
4.7 INTELLIGENT TUTORING SYSTEMS ... 46
4.8 DYNAMIC PRESENTATION ... 46
4.8.1 Examples of Dynamic Presentation: ... 47
4.9 SOFTWARE ARCHITECTURES ... 47
4.10 EVALUATION ... 51

5 SUMMARY AND DISCUSSION ... 52

5.1 PUTTING IT ALL TOGETHER - TWO POSSIBLE SCENARIOS ... 57
5.1.1 Gradual tool selection and de-selection - toolbox/workbench metaphor ... 57
5.1.2 Task driven tool selection - multiple workbench metaphor ... 58
5.2 RESEARCH CONTRIBUTIONS ... 59

6 REFERENCES ... 61

1 Introduction

End-user applications have changed dramatically since the introduction of the PC nearly two decades ago. The sharp increase in raw compute power has translated into applications with sophisticated graphical user interfaces and with considerably more functionality than their predecessors[1]. Look, for example, at word processing. In the early 1980s a package that included basic input, basic editing, basic page formatting, and the ability to save and print was probably thought to be comprehensive. For a word processor to be competitive today it must include some drawing and spreadsheet capabilities, the ability to import and export just about any format, the ability to format the document in a multitude of ways, and the list goes on. Despite significant efforts in the domain of Human Computer Interaction to make the interface more intuitive, our interfaces are in no way optimal [Raskin, 1997] and the functionality explosion has only furthered the complexity of software[2].

What about the user? As we move towards universal access it is timely to ask such questions as how users are currently managing complexity, whether complexity has become a barrier to effective and efficient use of human resources, and whether in fact the complexity of today's systems actually presupposes a sophisticated user? Have we perhaps reached the edge of accommodation and further advance will require a paradigm shift? Central to the issue of how users cope with complexity is how in fact users learn to use complex systems. What learning strategies do users employ? Is there a way that systems could be designed to better support the learning process? What can we learn from users' learning in other domains?

This paper is a literature review that addresses these questions and other issues involved in the learning of functionality-filled software that serves a diverse user population. In particular it covers two quite symmetrical approaches to user interface design: learning how users learn complex systems and designing accordingly, and having the system learn about the user and adjusting accordingly. The first approach is rooted in fundamental practices of the field of Human Computer Interaction. It involves achieving an understanding of how users learn by studying the users themselves through a number of well established techniques. Once the process of learning a complex system is better understood, then designs that support learning can be implemented and tested. The second approach is rooted more heavily in the field of Artificial Intelligence. It involves the system creating and maintaining a representation of the user's knowledge, habits, and needs. In accordance with this knowledge base, the system can perform some tasks (perhaps rudimentary or perhaps time consuming) on behalf of the user and can be adjusted to better accommodate the individual user. The goal with this approach is to place some of the burden of learning on the system.

---

[1] This trend of software expanding to take advantage of hardware capabilities is not unique to computing but has appeared in the evolution of many appliances, including food processors, digital watches, and video tape recorders [Card, 1989].

[2] It is worth noting that this trend towards complexity is not limited to software. It is evident in hardware [e.g., Myer and Sutherland, 1968] and is likely a general trend outside the realm of computing. Once a *system* becomes too complex and is no longer supportable, the general solution is to build specialized tools.

The purpose of this review is to identify research directions and future contributions in the area of learning complex computer systems. The bodies of literature reviewed include human-computer interaction, learning, intelligent and adaptive user interfaces, and even some trade literature.

The paper is divided into four main sections. The first section, *Setting the Stage*, highlights the literature in which functionality is addressed and explains some of the factors that have motivated the functionality explosion. The second section, *Learning to Use Complex Systems*, covers the first approach to user interface design by looking broadly at how people acquire skill in the context of computing environments. The third section, *Intelligent User Interfaces*, covers the second approach and describes the contribution of the Artificial Intelligence community towards reducing complexity in the interface and improving user performance. The final section provides a brief summary, discussion, and some possible future research directions. As an alternative path through the document, the reader may wish to skip directly to the final section after reading *Setting the Stage* in order to get an overview of the literature surveyed. With this alternative, the details in the third and fourth section can then be used for reference purposes.

2 Setting the Stage

2.1 Terminology

There is not yet a widely accepted terminology for the ***functionality explosion*** but it appears in the literature under the terms ***featurism*** [Constantine, 1995] and under the umbrella of the term ***bloat***[3] [Kaufman and Weed, 1998; Munk, 1996; Kesterton, 1998]. According to a workshop at CHI 98 entitled "Identifying Interface Bloat," bloat describes the perception of some users of a heavily-featured system. For some users, having a heavily-featured system is not problematic, in fact it is desirable. For other users, such a system evokes a negative response. These users find a system to be bloated when it is not obvious how to accomplish tasks, when there is more in the interface than the user wants to use, and simply when there is a lot to look at in the interface and these interface objects seem crowded and cluttered.

The term ***functionality*** is generally well understood and perhaps that is why authors neglect to define it precisely. Even in Goodwin's [1987] article on functionality and usability, the definition of functionality was assumed but not made explicit. For the purposes of this review functionality will be defined as: operations that modify a work object[4], operations that present a different view of the work object, operations that change the state of the

[3] Based on a conversation with Leah Kaufman, "bloat" originally appeared in the literature in reference to the impact on performance that resulted from adding new features, e.g., a bloated system was one that took too long to load or that required too much disk space.

[4] Work object is used to denote the artifact that the user is manipulating, e.g., a document, a presentation, a spreadsheet, an animation, an image, etc.

interface, operations that change settings in the interface. Although most functionality is accessible through interface objects[5] it is the operation itself that is considered to be the function. For example, the common operation to *cut* during editing is the removal of a selection from the work object and its placement on the system clipboard. Generally this operation is accessible through a button on an application's toolbar, through a menu item, and as well through the standard hotkey *Ctrl-X.* Despite the fact that there are multiple ways to access this operation, it only represents a single function. This latter example illustrates that as functionality is added, interface objects that access the functionality must also be added and this is not necessarily one-to-one because many functions have multiple accessors.

The term ***feature*** is more general than ***function***. It tends to refer to both the accessors of a function and the function itself. The perception of bloat can refer to both of these.

Throughout this review the adjective ***complex*** is used to describe technology. This term is relative and subject to interpretation. However, for the purposes of this paper, it can be taken to describe technology that has sufficiently many features that a person would not simply be able to walk up to and make substantial use of without investing significant learning time. A standard word processor provides one example of a complex system.

The classification of users into ***novice*** and ***expert*** categories is common in discussions about software skill. Unfortunately, the use of the term novice is not consistent in the literature. Novice is sometimes used to refer to someone relatively inexperienced with computers in general. It is also used to refer to someone who is new to a particular application [e.g., Nilsen et al., 1993]. Often it is not clear [e.g., van Oostendorp and Walbeehm, 1995] how novice is being used. A more defining terminology is needed because someone who is new to an application but who is a very experienced user is very different than a generally inexperienced user. Greenberg and Witten [1985] introduced the term *foreign users* for people who have no prior experience of a given system but are familiar with computers in general. This term has not been adopted by other researchers.

2.2 How are users coping?

Most computer users are constantly being faced with more on their desktop. It has been estimated that an office worker who relies primarily on three basic applications such as a word processor, a spreadsheet, and a graphics package, as well as an operating system, will be required to learn a software upgrade on the average of every six months [Franzke and Rieman, 1993]. With every upgrade, inevitably comes new functionality.

There has been minimal research to date that addresses how users are managing in the face of all this functionality. It doesn't appear as though anyone has taken on this issue directly. There are some accounts that are informal and qualitative in nature while others are more

[5] Interface object refers to a visible object that can be manipulated by the user. In GUI development toolkits these objects are called widgets and include: menus, toolbars, buttons, etc.

formal and include some quantitative measure. Here is an example of an informal account that likens featurism to a disease:

> "Word processors, and a growing legion of our most important software tools, have become victims of creeping featurism, a serious malady of user interfaces that strikes software in its prime and can, if left unchecked, cripple the user. Untreated, creeping featurism can leave users with an agoraphobic response to large, open dialogue boxes, or even with a lingering fear of unknown menus." [p. 162, Constantine, 1995]

Others have alluded to these complex systems as being "nightmarish" for the user:

> "Our present systems have come to be as large, complex, and nightmarish as the mainframes they first displaced (mainframes have become larger still; but most computer users don't have to deal directly with them on a daily basis)." [p. 99, Raskin, 1997]

Some survey research by Munk [1996] has suggested that having powerful PCs filled with heavily featured software can reduce users' productivity: "Too much horsepower on the desktop can have the perverse effect of cutting productivity." For example, she reported that in a survey of six thousand workers conducted by SBT Accounting Systems in San Rafael, California, it was found that users spent five hours a week 'futzing' with their PCs. It is not entirely clear what constitutes futzing, but she includes the following as big time wasters: waiting for programs to run, reports to print, repair men to show up, technical support folks to pick up the telephone, and organizing and clearing out cluttered disk storage.

Carroll and colleagues [Carroll and Carrithers, 1984a, 1984b; Carroll and Mack, 1984; Mack, Lewis, and Carroll, 1983] created what they called a *Training Wheels Interface* for a commercial word processor. This interface essentially blocked some functionality and error states, such that when users tried to select a blocked function, a message was displayed that indicated that the function was unavailable in the Training Wheels Interface. Two studies were conducted each with twelve novice users that compared the training wheels system (TW) to the complete system (CS). In the first study it was found that the TW users could complete a simple word processing task 21% faster and spent significantly less time ($p<0.05$) recovering from errors when using the altered system compared to the CS novices. *Error* is defined here as a departure from the create and print action path and *recovery* is defined as a subsequent return to that path. In a post-session comprehension test of word processor basics the TW group performed significantly better than the CS group ($p<0.05$), and in a questionnaire designed to reveal attitude toward work, the TW group scored significantly higher than the CS group ($p<0.05$). The authors conjecture that because the TW users were more successful, they may have felt better about themselves and about work in general. The second study was almost identical to the first and most of the mean differences in this study closely tracked those of the first study. Two differences in the second study were that the TW group committed significantly less errors ($p<0.005$) but there were no significant differences in the posttests on comprehension and work attitude.

Franzke and Rieman [1993] conducted a study with twelve users who had an average of two years experience with Macintosh computers. The study compared how long it would take users to learn how to create a default graph in two different versions of a graphing package. They found that it was significantly faster ($p<0.05$) using the earlier version of the package than the later version. The graphing task specifically required the navigation and completion of a particular dialog box that roughly tripled in features between the two versions tested. Clearly, if the experimental task had required the use of the features available only through the newer version's dialog box, then it would probably have been the case that using the newer version would be faster. (There would, of course, be no comparison at all if it was infeasible to accomplish the task in the earlier version.) This raises the question of how much of the functionality made available is actually used.

Franzke [1995] investigated the impact of the number of interface objects on a user's ability to find the appropriate object. She conducted an experiment with seventy-six reasonably experienced users (average of just under three years of Mac experience and average familiarity with three Mac applications) that included two trials. It was found that as more objects were displayed at a given time, it took significantly longer for the users to locate the object needed. But this main effect interacted with trial. Thus the action times during the first trial (the exploration trial) were more affected than those during the second trial. During the second trial, the number of objects did not matter as much. There was a three-way interaction between the number of objects, the quality of an object's label, and the trial such that the search time for poorly-labeled objects was significantly worse when there were many objects to search in early trials. If the label quality was good there was no effect of the number of objects on display. There was another triple interaction observed between the type of interaction, the number of objects on display, and the trial. If there were many objects to search, action times during trial one were inflated especially for interactions that were difficult to discover. Such interactions refer to, for example, clicking on a dialog-box button, editing text, using a radio button, single clicking on a graph object, etc. In particular, the types of interactions that fall under the category of direct manipulation required more time.

In terms of total functionality usage, Nilsen et al. [1993] conducted a longitudinal study of people using a commercial spreadsheet package. In general, their research found that people only use a subset of an application's functionality and they don't often master even this subset, let alone learn and master all the functionality. Kesterton [1998] reports that on average people use only about 13 percent of the computer features and programs that they own. The Nilsen et al. study lasted 16 months and included twenty-six subjects who were followed during their learning of Lotus 1-2-3. The subjects were all new MBA students who were expected to use Lotus 1-2-3 for school related work. Each subject was "tested" (in a laboratory setting) at the outset and then was subsequently given an almost identical test at each of three equally spaced intervals, so there were four testing sessions. The tests consisted of tasks that required knowledge of basic Lotus 1-2-3 functionality. The results showed that fourteen of the twenty-six subjects were able to complete all the tasks in the test, seven were able to complete most tasks but made some errors, and the remaining five subjects committed multiple errors which displayed a basic lack of knowledge about Lotus 1-2-3.

Nilsen et al. [1993] reviewed the HCI literature and found that studies of the growth of software skill show "that people attempt to fully master the current task-related skill before moving on to more complex, advanced skill or those relevant to other tasks." This suggests that users prefer to feel like an expert in a functionality subset rather than a novice in the total functionality. When applications do give users the choice to operate at different expertise levels within a complex system, Shneiderman [1997b] claims that users are content remaining experts at level 1, rather than dealing with the uncertainties of higher levels. The only systems he cites as giving this choice are computer/video games and HyperCard. (Although most systems permit tailoring, he is specifically referring to systems that allow the user to set a level and the subsequent availability of functionality is strictly based on the level.)

2.3 Why featurism? Why the complexity?

The documented accounts and studies found in the literature suggest that a lot of functionality is unused and that unneeded functionality is distracting to users. This may indicate that a return to simpler systems would be beneficial. If we assume that the average user would make better use of less-featured software, then why has most software become more-featured? The literature suggests a number of reasons, namely, that features are needed in order to market the product successfully, that it is the programmers themselves (i.e., the technically-literate) who are deciding which functionality to include and are designing how the functionality is included, that the evolutionary process of software development does not easily accommodate global redesign, that additional features are needed in order for an application to integrate with other applications, and that usability guidelines favour giving users multiple ways to perform the same action. Each of these is discussed briefly below.

Marketing is driving featurism because features sell [Constantine, 1995]. It is somewhat a dichotomy; despite the fact that most buyers will never use many of the options, it is a comfort for these same buyers to know that the options are there just in case they may be needed someday. This can also be viewed as consumerism, i.e., that users want more for their money regardless of whether or not it will be used [Kaufman and Weed, 1998]. In the trade press, software reviewers clearly focus on features by using tidy comparison tables that are packed full of different markers (checks, dashes, circles with various fill) which denote the extent to which a package has a certain capability. Vendors attempt to have the most checks (or filled circles) on the function list and consumers learn to discriminate between full-featured applications and those with less features.

The request for new features comes primarily from experienced users and these features are supplemented, designed and implemented by programmers who are also experienced users

[Computer Science and Telecommunications Board, 1997]. It is sometimes the case that programmers want to add easily coded features with little concern for the extent to which the features will actually be used. The mentality is that if it is easy to code, then the cost is low and so even if it only benefits a few, it is worth it. In addition, programmers are creative people and may want to add a new feature because it is innovative or challenging to code [Kaufman and Weed, 1998].

It has been suggested that it is not the addition of features that has caused the complexity but rather the manner in which they have been added. The process of software engineering is evolutionary and rather than maintaining a clear and consistent global design through versions of an application, it is more often the case that the software turns into a patchwork of parts and pieces and consequently usability suffers greatly:

> "Creeping featurism results from the slow accretion of capabilities and is reflected in a bumpy and irregular user interface marred by idiosyncrasies and special functions that seem to grow like warts or carbuncles in the oddest places." [p. 163, Constantine, 1995]

Additional features appear in software products for compatibility or integration purposes. This is exemplified in product suites such as Microsoft Office, in which the user is given the ability to load a file from one Office product into another [Kaufman and Weed, 1998].

Lastly, new features may appear for what is considered to be a usability reason. It is sometimes thought that if users are given multiple ways to do the same thing, that the usability of the system is enhanced [Kaufman and Weed, 1998].

To summarize, the functionality explosion that has taken place is largely attributable to marketing forces, and the resulting features are primarily targeted at and designed for the sophisticated user. Those users who are average or not even average use a minimal amount of the total functionality and preliminary research suggests that these users cope better with simpler versions of software. This conclusion does not bode well for the goal of universal access.

2.4 The Goal of Universal Access

Both in Canada and in the United States the goal to provide universal access to computing technology has been set. Industry Canada is targeting to make Canada the most connected country in the world by the year 2000. The Schoolnet program is just one example of this initiative. Through this program all 16,500 schools and 3,400 public libraries across Canada will acquire Internet connections by the end of 1998 [Manley, 1998]. The United States shares Canada's goal of becoming a connected country but recognizes that connections alone will not guarantee access to all. The U.S. National Academy of Sciences has accepted the challenge of making the Nation's Information Infrastructure (NII) accessible and usable by all. In a workshop held to address this challenge, the attendees decided that the interfaces to the computing and communications systems that dominate the NII should be referred to as *every-citizen interfaces*. This term reflects the project's mission to examine the necessary requirements for every citizen to be able to use the resources available through the NII. The attendees recognized that even citizens without disabilities can struggle with information technology:

> "...even though the usability of systems has improved substantially over many years, current interfaces still exclude many people from effective NII access. Most obvious are individuals with physical and other disabilities, but as articles in even the national and business press attest, people without such distinguishing characteristics, even expert users of NII systems, experience difficulties that constrain or even preclude their full use of NII resources." [p.1, Computer Science and Telecommunications Board, 1997].

The following are characteristics that were determined to be desirable for effective every-citizen interfaces: easy to understand, easy to learn, error tolerant, flexible and adaptable, appropriate and effective for the task, powerful and efficient, inexpensive, portable, compatible, intelligent, supportive of social and group interactions, trustworthy (secure, private, safe, and reliable), information centered, and pleasant to use.

2.5 Research Direction

The apparent frustration by at least some of the non-expert users with complex software suggests that the interface needs to be reconceptualized to meet the needs of all users, regardless of skill level. It seems intuitive that an interface offering the user only the functionality pertinent to his/her skills and needs would be easier to use than one that offered functionality beyond his/her comprehension [Cote-Muñoz, 1993]. Imagine a system that knows the user's skill, needs and current task, and then presents exactly the appropriate functionality. When these skills, needs, and task change, the interface would adapt easily and present the functionality required to match these changes. Clearly the system just described is unachievable. It would require the system to accurately predict all changes in both user skill and needs. Just because the ideal is not achievable, however, does not mean that we must stay with the current paradigm.

3 Learning to Use Complex Systems

In order to understand how featurism impacts the learner it is necessary to first step back and ask how, in general, people learn to use a complex system. It is instructive to first identify why learning is necessary at all.

Bösser [1987] situates the role of learning complex systems as shown in Figure 1 which has been adapted from Baecker and Buxton [1987]. The *prescriptive model of the system* represents the set of skills that are required for a user to successfully use a system fully. The *descriptive model of the user* represents the set of skills that a user has when first encountering the system. It is most often the case that the descriptive model is a proper subset of the prescriptive model and is thus shown as such in the figure. The goal is for the prescriptive model to equal the descriptive model. To achieve this goal, the gap between the required and actual skills (i.e., the set difference) needs to be bridged. The bridging can occur through better *design* which reduces the required skill or by *learning* of skills on the part of the user. Thus, Bösser notes that good design and learning are compensatory. Weaknesses of the system can be overcome by learning, and effective design reduces the burden on the user's cognitive system.

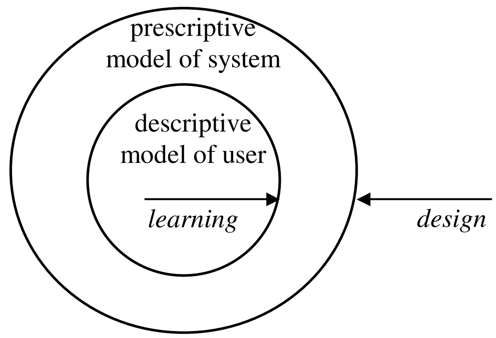

Figure 1: learning and design can be used to bridge the gap between the descriptive model of the user and the prescriptive model of the system [Baecker and Buxton, 1987].

There are potentially two things to be learned by the user: the task and the interface. Bösser [1987] refers to these as the *task* and *tool knowledge* that need to be acquired. Baecker and Buxton [1987] distinguish these as the *functional* and *operational* problems to be solved. Operational problems have to do with the means of performing work whereas functional problems have to do with the content of that work. They note that one objective of user interface design is to minimize the need for operational problem solving because all of the cognitive resources consumed at this level are being diverted from completing the task that was the reason the computer was adopted in the first place.

The learning being addressed in this review is primarily that of learning the interface, or in the terminology above, how users overcome the operational problems. Although users may not have perfect task domain knowledge before using an application that supports a domain, it must be assumed that users must have some understanding of the task to be able to use the tool. For example, if someone doesn't know what the alphabet is and doesn't understand words or paragraphs, then regardless of how well a word processor is designed, it is unlikely that the user will be able to learn how to use it (for its intended purpose anyway). The question of how to design a system to support the learning of task knowledge is an interesting one and is the primary focus of researchers investigating Learner-Centered Design which is described in Section 3.5.3.

3.1 What is learning?

Bösser [1987] notes in his literature review "Learning in Man-Computer Interaction" that in his experience finding a general definition for learning wasn't possible. In some schools of empirical psychology, learning has been associated with *behaviourism* and a representative definition would be: "learning is a change in behaviour occurring as a result of experience." This definition stresses observable variables and is therefore only meaningful when the procedures for experimentally manipulating 'experience' and for observing the behaviours that change as a consequence of these manipulations are defined.

Bösser [1987] notes that from a cognitive point of view, this type of definition is too simplistic and that there are cognitive processes for which experimental procedures cannot be defined. He cites the following non-operational definitions which he suggests are almost meaningless for empirical psychology:

- "Learning is making useful changes to the mind" [Minsky, 1985 cited in Bösser, 1987]
- "Learning is constructing or modifying representations of what is being experienced" [Michalski, 1986 cited in Bösser, 1987]

Current cognitive theory emphasizes three interrelated aspects of learning [Resnick, 1990].
1. Learning is a process of knowledge construction, not of knowledge recording or absorption.
2. Learning is knowledge-dependent; people use current knowledge to construct new knowledge.
3. Learning is highly tuned to the situation in which it takes place.

Learning has also been described as the acquisition of knowledge where knowledge can be both factual knowledge as well as procedural knowledge. Skill is procedural knowledge that has been optimized for execution speed and accuracy through practice. Unskilled behaviour is problem solving [Card, Moran, and Newell, 1983 in Baecker and Buxton, 1987] which requires the user's attention and uses a relatively large number of resources [Baecker and Buxton, 1987]. These definitions of skilled and unskilled behaviour are consistent with cognitive psychology where skill acquisition is defined as the process of compiling and proceduralizing individual steps. Baecker and Buxton have likened this to a *cognitive subroutine.*

With respect to learning a system, we encounter in the trade literature, in some of the HCI literature, and certainly in the popular press the common buzzwords "ease of use" and "user-friendly." When a system provides ease of use or a system is user-friendly it implies that a system is easy to learn but these are all in fact vague concepts [Baecker and Buxton, 1987; Bösser, 1987]. Baecker and Buxton suggest that a clearer formulation of the goal that is intended by these terms is the goal of accelerating the process whereby novices begin to perform like experts. (See Section 2.1 for discussion on user categorization.) Nilsen et al. [1993] observe that the HCI literature adopts skill acquisition as representative of learning and that the psychological literature focuses on the improvement in performance, where performance refers to the speed and accuracy at which tasks are completed.

3.2 External Factors Affecting Learning

People do not simply respond as automatons to given stimuli. People are different and the contexts within which people learn to use computers and computer applications are different.

3.2.1 Motivation

Bösser [1987] notes that motivation plays a significant role in learning. Humans execute actions based on multiple goals and constraints, some of which may be conflicting. He says that the goal of user behaviour is to maximize a multi-dimensional utility criterion, that it is important to consider the relative utility associated with the reachable goal conditions, and that motivation guides behaviour and therefore also guides learning based on practice (i.e., determines which skills will develop). Motivational factors associated with human-computer interaction are largely determined by the task and are specific to the user's current context. Time management is considered to be the most important factor. The loss of current work time is associated with high negative utility. Other likely factors include: error avoidance, improvement of performance or quality of product, building confidence in tool usage, improvement of qualification, and reduction of stress by automation of skill. In his review, Bösser urges Human Factors practitioners to identify the dimensions of utility and the constraints which are of relevance for the user in specific work situations. From this we can conclude that enumerating the motivational factors and quantifying them is an on-going area of research.

3.2.2 Individual Characteristics

Unlike motivation, which often varies not only between users but also within users depending on the task and other constraints, characteristics such as cognitive ability and cognitive style are inherently individual characteristics. Olfman [1987, in Davis and Bostrom, 1992] reviewed the role of individual characteristics and their impact on users learning new software. He found that cognitive ability (memory ability, reading/semantics ability, and visual ability), cognitive style (analytic/heuristic, learning style, and preferred learning mode), computer experience (with specific software applications), and a number of other traits (e.g., age, grade point average, mathematics ability) did impact learning.

Davis and Bostrom [1992] conducted a study that compared the impact of two learner characteristics, individual learning mode and visual ability, on the ability to learn both a direct manipulation system and a command-line system. Their hypotheses regarding the impact of being an abstract versus a concrete learner were not supported by the study. With respect to visual ability, however, the hypothesis that high visual users would perform better overall than low visual users was supported. High visual users also tended to perceive the systems under study as easier to use than did the low visual users. One must be cautious not to over generalize the results of a study such as this. The phenomena under study here are reasonably complex and thus multiple studies using different methodological approaches are likely required to gain an understanding of their true impact [see McGrath, 1994]

3.3 How does one learn?

Learning has been identified as a means of matching the user's skills to the skills required by the interface. The literature documents a number of different approaches to learning a complex system which include exploratory learning, learning through transfer of prior knowledge, formal training, learning through user support provided with a system (documentation, online help, tutorials, animations, and demonstrations), and learning through assistance from colleagues and friends. Exploratory learning, as will be discussed below, is the most common approach to skill advancement and so although each of these methods is covered in this review, exploratory learning is given the greatest coverage.

Discussions about Computer Aided Instruction and Intelligent Tutoring Systems have not been included. The purpose of these systems is to help the user learn, however, they are generally geared towards facilitating task learning (functional learning) and not learning the interface (operational learning).

3.3.1 Exploratory Learning

Learning by doing [Carroll and Mack, 1984], *learning by trial and error*, *learning in context* [Rieman, 1996], and *active learning* [Kerr and Payne, 1994] can all have slightly different meanings than exploratory learning, however, they generally all represent the same phenomenon which is acquiring knowledge by navigating through a system, trying different actions, and assessing the results of those actions. There is no general principle of how exploratory learning works and most of the literature on exploratory learning is domain-specific (conversation with Dan Keating, chair of Human Development and Applied Psychology at Ontario Institute for Studies in Education). Thus the literature documented here is specific to learning to use technology.

The evidence regarding the efficiency of exploratory learning as a means to acquire knowledge is conflicting [Carroll and Mack, 1984]. Bösser [1987] compares learning by doing to formal training and notes that although the latter may take more time, it results in more productive work afterwards. Kamouri et al. [1986, cited in Trudel and Payne, 1995] conducted a study that compared exploration-based and instruction-based techniques (practice-oriented, tutorial examples in a manual) for learning a computer-simulated device. Two days after training, subjects who had learned by exploring were more successful at transferring procedures to a novel, analogous device. The explanation given for this result by the authors was that exploration encouraged the development of abstract representations and analogical reasoning while the more passive learning that resulted from following textual instructions did not.

There is reasonable evidence in the literature that exploratory learning is used more often than any other method [Howes and Payne, 1990; Rieman, 1996; Carroll and Mack, 1984].

Howes and Payne [1990] note that exploratory learning is an everyday reality - learners spend a lot of time exploring, whether out of choice or out of necessity. They cite the following reasons: no documentation is available, it is too much trouble to read documentation even if it is available, there is an urgent need to accomplish particular practical tasks, and exploration is fun.

Rieman [1996] conducted a field study to address the question: Within the work and computing environments currently available, when and how do users learn to use the software they need for their daily tasks? He found that users predominantly use exploratory learning and they do so in the context of real tasks. Users are very concerned about accomplishing the tasks required for their job and thus prefer a just-in-time, task-driven approach to learning. Rieman uses a broad definition of exploratory learning: learning "through trial and error, through interaction with other users, through occasional reference to manuals if they are available." It is not surprising that his definition is less constrained than most definitions because it is derived from a field study and not a controlled experiment. Others have used narrower definitions. For example, van Oostendorp and Walbeehm [1995] define exploratory learning to be "a special case of problem-solving that consists of a search process in a basic problem space, while the learner has no, or almost no, specific domain knowledge."

Even though exploration is often used for learning an interface, research shows that when it is used in its most basic form it doesn't result in effective learning. Trudel and Payne [1995] found that unfettered exploratory learning is rather unreflective and unsystematic. They suggest that if users are left to explore in an unrestricted fashion, they do not behave adaptively, i.e., they interact too much, and think too little. Further, they suggest that this maladaptive behaviour is actually encouraged by the interactivity of modern computer-based devices, i.e., by the ease of making actions and by the rapid visual feedback that is derived[6]. Payne and Howes [1992] identify three exploration traps:

- Learners may achieve the desired effects but forget the sequence of actions taken to accomplish the success.
- Learners may remember a sequence of actions but it may be a suboptimal method.
- Learners may inappropriately characterize their accomplishments, or inappropriately parse device transformations which makes generalization to different situations problematic.

Trudel and Payne [1995] demonstrate that increasing the cost of interaction at the interface can improve performance and learning. They argue that when *goal management* and

[6] This high "interactivity/poor learning" versus "reflection driven interaction/better learning" can be seen as a form of speed/accuracy tradeoff. If the objective of a task is speed, then accuracy is generally sacrificed. On the other hand, if the objective is accuracy, then speed is sacrificed. This type of tradeoff is hardly limited to interacting with computers.

reflection are included with exploratory learning, more is learned. They conducted an experiment with subjects learning to use a computer-simulated digital watch in which reflection was influenced by the imposition of a keystroke limit (i.e., a limit on the amount of physical interaction with the device) and goal management was influenced both by providing subjects with lists of goals and by limiting subjects to explore one part of the device at a time. They found that imposing a keystroke limit did lead to improved exploratory learning: despite exploring the device for less time, the subjects who had limited keystrokes demonstrated both superior declarative and procedural knowledge. One interpretation of this result is that by limiting the keystrokes, each interaction with the device is a scarce and valuable resource which encourages subjects to pay more attention to, and think harder about the results of the interaction. A similar result was found when subjects were forced to explore one mode of the digital watch at a time. The goal list manipulation also improved exploration but to a lesser degree than the other two manipulations. This result seems to suggest that goal-oriented exploration, while important, is not as significant a determinant for effective learning as are reflection on interaction and constraining the exploration space.

Trudel and Payne [1995] note that their findings pose a dilemma for the designers of interactive systems because one of the main appeals of such systems is their explorability. They suggest that user interfaces could be augmented with special "exploration facilities" which compromise some of the usability of standard operation or add information to the system display so as to enhance exploratory learning. They note that the training wheels interface is one example of this.

Research conducted by Svendsen [1991] suggests that another way to force reflection is to make the interface more difficult to use. When he compared users doing the same task with a direct-manipulation interface to users using a command interface he found that those who used the command interface demonstrated superior problem solving ability. He concluded that systems that are considered to be user-friendly can in some cases reduce the users' problem-solving ability. Despite this, the users in his study showed a clear preference for using the direct manipulation interface. This dichotomy poses a challenge for designing for learnability.

Payne and Howes [1992] developed the task-action trace to support informed reflection during exploratory learning. It is supposed to rescue the user from the 'how did I do that?' trap. Through dialog monitoring the task-action trace provides: a dynamic trace of the user's actions; a parallel trace of the tasks the user accomplishes with a display of the mapping from actions onto tasks; and a filter of the user's actions for any particular task, displaying only the necessary subset. These traces are maintained in a window separate from the application window. Payne and Howes ran an informal study of the task-action trace which suggested that some users find the trace sufficiently helpful to interact with it repeatedly during the first four hours of learning. They also found that presenting learners with an additional "thing" to learn can be problematic. Pilot subjects complained that being presented with special instructions for the trace alongside the main learning task was overwhelming. It was concluded that learning support tools should not burden their users with new learning demands, or they will very likely not be utilized.

The research by John Carroll and colleagues in the 1980s [Carroll and Carrithers, 1984a, 1984b; Carroll and Mack, 1984] on the Training Wheels interface took a different approach to that of the task-action trace. Rather than adding supporting information to guide exploration, the Training Wheels interface blocked off error states and functionality in a word processor that was not needed for the simple task of typing, saving, and printing a document. With this blocking, novice users were able to accomplish the task significantly faster and spent significantly less time recovering from errors than did novice users who used the full version of the word processor. Carroll and Carritthers [1984b] identify three main aspects of the design of the Training Wheels system: it limits the punishment associated with making errors in learning; the user errors elicit feedback that can help learners discriminate learning targets from errors; and it limits and focuses the potential space of user options in the learning situation. The first two aspects given, designing for error and providing feedback, are now considered integral to good design.

Franzke and Rieman [1993] investigated whether earlier versions of a software package would provide a "natural training wheels environment" for later, more complex upgrades. The experiments conducted confirmed this hypothesis for the software tested. It was found that subjects actually spent less total time performing the same task twice, once in the earlier version of a graphing package and then again in a later version of the same package, than other subjects who performed the task only once in the later version with no preparation. Franzke and Rieman note that these results suggest that earlier versions of software could be useful for training, either in a formal training program or as production software for novice users. The success of either approach would be highly dependent on both the users' tasks and the differences between versions of the software.

Franzke [1995] conducted a study to determine how explorability is impacted by certain interface conditions. In particular, she looked at whether the ability to discover the appropriate interface object is dependent on the number of distracting interface objects, on the type of interaction required, and on the goodness of a label. For reasonably experienced users it was found that the more objects that were displayed at any given time, the longer the action times to access the correct object. A similar finding was the case for the quality of the label. This finding, however, interacted with the trial, which means that in subsequent uses of the system, the impact of the number of objects and the label quality was lessened. It was also found that if there were many objects to search, action times during the first trial were inflated, especially for interactions that were difficult to discover, such as direct manipulation interactions.

To summarize, research shows that learning through exploration is the preferred method although it may not necessarily be the most efficient or effective method of learning how to use a system. It is precisely the interactivity of applications that encourages exploratory learning, but it is also the interactivity that allows for unproductive exploration and thus poor learning. Exploratory learning can be enhanced when the learner is forced to reflect on his/her interactions and when the exploration space is constrained. Reflection can be forced by limiting the amount of interaction the user can have with a system or by making the interaction more costly.

3.3.2 Learning Through Transfer of Prior Knowledge

Learning through transfer of prior knowledge has been called *learning by knowing* and *metaphor* [Carroll and Mack, 1984]. It is also sometimes referred to as *transfer of learning*, which Bösser [1987] defines as "the saving in learning a second task caused by the previous learning of a similar task."

The following questions arise: Where is knowledge acquired? Once acquired, will knowledge always be available? For the purposes of human-computer interaction Bösser [1987] distinguishes *task* and *tool* knowledge, either of which may be *general* or *specific.* Task refers to domain knowledge and tool refers to knowledge about the system or device. (Tool knowledge was earlier referred to as interface knowledge and operational knowledge in the first part of Section 3 in this document.) Bösser defines sources of knowledge and the length of time that knowledge is maintained as given in Tables 1 and 2, which are adapted from his book.

	Knowledge of Task	**Knowledge of Tool**
General	formal education	education and training
Specific	vocational training	tool-specific training, practice

Table 1: Sources from which knowledge is obtained [Bösser, 1987].

	Knowledge of Task	**Knowledge of Tool**
General	the rest of your life	decades or more
Specific	decades or more	months to decades

Table 2: Duration that knowledge is maintained [Bösser, 1987].

The type of knowledge that is most relevant to HCI is tool-specific knowledge, which as shown in Table 2 can potentially be very short lived. This type of knowledge must be acquired with each new tool even if there is transfer of task knowledge and general tool knowledge. Bösser [1987] notes that general and specific knowledge from other domains may be prerequisites for acquiring tool-specific knowledge and, in particular, extensive knowledge of the task domain in which the tool is used is often a precondition. As explained at the beginning of Section 3, the need for domain knowledge can be seen even by looking at basic word processing. If a user doesn't understand the alphabet let alone the meaning of a word, a sentence, or a paragraph, it is highly unlikely that the design of the word processor is going to impact that user's ability to create a document.

Carroll and Mack [1984] acknowledge that prior knowledge can be problematic. They studied how users learned to use a word processor and how the typewriter metaphor was prevalent. They note that difficulty arose when the metaphor did not fully hold. For example, a number of confusions using the word processor were experienced when the learners used the backspace key which not only moved the cursor back one space but also erased the character and, similarly, using the spacebar with the cursor positioned over text not only moved the cursor forward but also introduced a blank space and shifted the text following the cursor by the width of the blank space.

It is natural for learners to rely on their prior knowledge. In fact, it is probably very difficult for learners to ignore what they know. And so it seems obvious that designers should design systems to account for users' prior knowledge. The difficulty with this design solution is that new technologies will necessarily be constrained by old technologies [Carroll and Mack, 1984]. Some key questions for future research might be: To what extent can we constrain new technologies by old ones? Is there an efficient method for making users aware of instances where the metaphor doesn't hold?

Broad questions regarding knowledge transfer and the domain of computer usage include: What kind of general education and training are necessary, desirable or optimal preparation for using computers? How does the knowledge acquired when working with one computer system transfer to working with a different system?

3.3.3 Formal Training

Formal training refers to learning through educational courses conducted by an instructor.

There are obvious advantages of formal training: students' misconceptions can be detected by the instructor, instructors can reveal functionality that students may not have found on their own, and efficient methods for accomplishing tasks can be made evident. The premise is that this training will result in better long term performance than other less structured methods of learning.

One disadvantage of formal training is that the content of a course is not often tailored to the learner's specific needs. The extensive principles that are covered in courses can be unnecessary for most users who execute only limited tasks. Another disadvantage is that users forget knowledge acquired during training when they have no opportunity to immediately apply it in their real work domain [Bösser, 1987].

Although the literature suggests measures to estimate the cost of training (fees for training course, cost of personnel during training, loss of work time during training, etc.), it doesn't suggest accurate measures for estimating the benefits of training [Bösser, 1987]. Figure 2 provides a visual representation for some aspects of the cost of training.

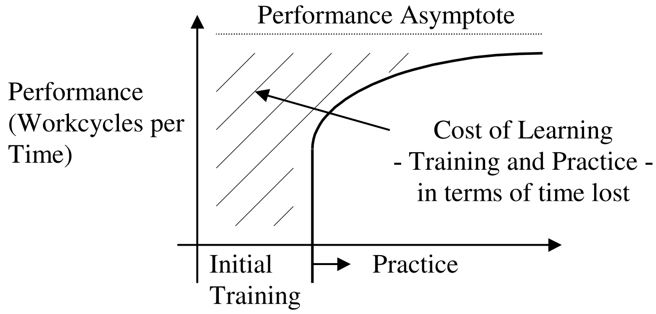

Figure 2: The cost of learning to use a device is the sum of performance lost for training and reduced performance during practice. Not only time lost, but also reduced quality of output may have to be considered as a cost factor [Bösser, 1987].

3.3.4 User Support - Online Help, Tutorials, Demonstrations and Animations, and Documentation

Providing both procedural and factual information through additional learning materials is one way to transfer tool and perhaps task knowledge to the user. For the purposes of this review, learning materials are categorized into online help, documentation and tutorials. This is not a perfect categorization and there is some overlap. Documentation refers to manuals and other printed materials. Online help generally contains similar content to documentation but may be structured differently and provides different affordances. Online tutorials and demonstrations are in some sense a subset of online help but refer to interactive or multimedia materials.

One problem common to supporting material, whether it be online or printed, is what is sometimes referred to as the "language problem." This problem occurs when the language used to describe concepts/functionality in supporting material does not match the language users use to describe the same concepts/functionality. This makes it very difficult for users to seek help on how to do something. The problem stems largely from the fact that it has traditionally been expert users who have been responsible for writing the documentation. Expert users have developed a rich vocabulary in the process of gaining expertise and they either forget that novice users will not have this same vocabulary or they are unable to anticipate the vocabulary that novice users might use and, therefore, simply use the vocabulary with which they are familiar. Borenstein [1985] found that the quality of help texts is more important than the mechanisms by which the texts are accessed and concluded that resources should be concentrated on technical writing rather than elaborate help mechanisms.

3.3.4.1 Online Help

Standard online help generally provides the same information that printed documentation contains. Examples are utilities such as the command-oriented *man* on UNIX and the hypertext-style help in PC and Macintosh systems. The most common form of online help is a list of article titles and an index of terms that lead to the articles [Shneiderman, 1997a].

Rieman's field study [1996] of 14 interactive computer users, who represented a wide range of computer skill and job duties, found that in general online help was inferior to manuals. He notes that this was a paradoxical result given that users' learning activities are task-oriented and time-constrained. Online help is the only learning aid that has some access to the user's current context and is the most readily available, especially in site-licensed environments and for laptop users. He suggests the following explanations for this paradox: the help window obscures the task window which inhibits users from using trial and error at the same time as online help; and the users' time constraint discourages use of reading and navigating online material, which is perceived to be slower than printed documents because of the reduced readability of online text [Gould et al., 1987, in Rieman, 1996].

Sellen and Nicol [1990] identify the problem that online help is often difficult to use, not that helpful, and often leaves users feeling ineffective and frustrated. In their user study at Apple Computer, Inc., they found that there are five different types of questions for which users seek help:

1. **Goal Oriented:** What kinds of things can I do with this program?
2. **Descriptive:** What is this? What does this do?
3. **Procedural:** How do I do this?
4. **Interpretive:** Why did that happen? What does this mean?
5. **Navigational:** Where am I?

Baecker and Small [1990 in Baecker, Small, and Mander, 1991] modified and extended these question types to the following:

1. **Identification:** What is this?
2. **Transition:** Where have I come from and gone to?
3. **Orientation:** Where am I?
4. **Choice:** What can I do now?
5. **Demonstration:** What can I do with this?
6. **Explanation:** How do I do this?
7. **Feedback:** What is happening?
8. **History:** What have I done?
9. **Interpretation:** Why did that happen?
10. **Guidance:** What should I do now?

Sellen and Nicol [1990], from their user study, developed the following five principles for the design of online help:
1. Online help should never be a substitute for good interface design.
2. Help should be context-sensitive; it should not take the user away from the task at hand.
3. Help systems should assist users in framing their questions and provide different help for different questions.

4. Help systems should be dynamic and responsive.
5. User shouldn't need help to get help.

Baecker, Grudin, Buxton, and Greenberg [1995] provide an optimistic view. They suggest that many of the early studies on online help which found it to be ineffective were based on old interfaces and screen technologies. They suggest that online help has traditionally received few resources but that at least some vendors are starting to pay serious attention to training, documentation, and online help.

There are *intelligent help* systems which attempt to tailor the help provided to the user's current context and skill level and in some cases provide natural language queries. More detail about this is provided in Section 4.

3.3.4.2 Tutorials, Demonstrations and Animations

Demonstrations and animations typically present essential system concepts and are thereby an effective way of introducing novice users to the system. Tutorial exercises guide the user through system concepts, giving the user an opportunity to try out the concepts.

Baecker, Small, and Mander [1991] investigated animated icons as a means of addressing the problem that users are often unsure of what an icon represents. The results from their user study showed that there was a significant benefit from the animations. Animations that were kept simple both visually and conceptually helped users clarify the purpose and the functionality of the icons.

Payne, Chesworth, and Hill [1992] investigated the instructional potential of what they termed a *pure* version of animated demonstrations. These versions had no commentary or supporting documentation for the animation. Such animation is analogous to what is seen in video games when a game is not in use. Their study found that a 2.5 minute animation, silent video of the MacDraw screen in use resulted in an almost 50% reduction in task completion time. The animation was shown to the users prior to their use of MacDraw.

Kerr and Payne [1994] compared the instructional efficacy of using animated demonstrations in both active and passive learning environments to teach basic spreadsheet skills. Active learning is characterized by problem-solving behaviour whereas passive learning is characterized by following a prescribed script which is common in commercial tutorials. The latter has been labeled a *scenario machine* [Carroll and Kay, 1988 cited in Kerr and Payne, 1994]. Results indicate a clear learning advantage of problem-solving, over prompted interaction (the scenario machine). It was found that animations played two key roles. The first is that simply watching animations provides a useful introduction into complex interfaces. The second is that animations can be effective as an *example following* resource for active problem-solving.

3.3.4.3 Documentation

Carroll [1990] recommends minimalist training materials because they support exploration. He found, for example, that short incomplete manuals can be more effective than the full "systems-style" versions of manuals.

Wright [1983] performed a review of documentation design. She found that common activities which all readers undertake are searching for information relevant to their present needs, understanding the information found, and applying the understanding gained. Wright described the functions that documentation needs to serve as ranging from tutorials to quick references to detailed explanations [1988, cited in Rieman, 1996]. In general, she advocates a "user-oriented" approach to the creation of documentation [Baecker and Buxton, 1987].

In Rieman's [1996] field study about how users learned, it was found that for task-oriented problem solving, the informants would typically use the documentation in addition to trying things on the system. He notes that this is an approach consistent with Carroll's minimal manuals. A somewhat unexpected finding in Rieman's study was that some of the informants relied on third-party manuals. (An example of such manuals is the series "*X* for Dummies" where *X* is anything from computer applications to golfing.) He notes that this is perhaps because of the limited number of "official" manuals that are available in a site-licensed situation. But he also suggests that this may be a means of overcoming the language problem. The hypothesis is that the informants may have tried trial and error and given up with the official manuals and online help and are using the third-party manuals because of the alternative language used and views given. The order in which the material is presented in alternative manuals may also be better matched to users' tasks.

Bösser [1987] reviewed the literature on documentation and found it to be inconclusive but did provide the following recommendations:

- Advance organizers are helpful for chapters and sections. An advance organizer refers to the presentation of instructional material such that a rough overview is presented first and is then filled in by the material presented later.
- Graphic display of the structure of menu hierarchies and command language grammar is helpful.
- The provision of both task and tool-oriented indices is needed.

3.3.5 Learning from Friends and Colleagues

A common way of furthering knowledge is to ask a friend or colleague for assistance. This type of assistance can take many different forms which include: asking informal questions to the closest person within earshot, posting questions to Usenet newsgroups or bulletin boards, sending a question via e-mail to a friend or system administrator, or watching over a colleague's shoulder as he/she presents a demonstration.

Rieman [1996] found in his field study that users frequently asked for assistance in person or by phone and often did so in combination with trial-and-error strategies.

Clement [1993] reported that when desktop computers were introduced into a University administrative office, the secretaries found that they learned more by studying on their own and by discussing difficulties with co-workers than what they had learned through formal courses and external sources of expertise.

3.4 Modelling How Users Learn

The goal of *cognitive modelling* is to model how humans gain knowledge and it does so within the framework of the human as information processor. This information processing paradigm rests on the assumption that human behaviour can be described in terms of information processes, essentially programs. Learning is one aspect of cognition and thus cognitive models that represent learning appear in the literature. One of the goals of cognitive modelling is to predict human performance. As a caveat, Preece [1994] notes that the cognitive perspective of the individual user performing various tasks at the interface is losing its recognition as an adequate conceptual framework for HCI. The traditional cognitive approach has neglected how people work in the real world, for example, it doesn't account for people interacting with others.

Rieman, Young, and Howes [1996] describe a cognitive model of exploratory learning that covers both trial-and-error and instruction-taking activities. One key aspect captured in this model is what they refer to as iteratively deepening attention. When trying to accomplish a task a user will normally use a label-following strategy which means that the user attempts to find a label that matches a word or concept in the task description. If such a label is not found the user will often repeatedly scan pull down menus or a subset of them with increasingly greater attention to each item. Rieman, Young, and Howes explain this behaviour in terms of dual search spaces: the application interface and the user's internal knowledge. Both of these must be explored in such a way that the costs and benefits are considered. The model implements this dual-space search by alternating between external scanning and internal comprehension thereby narrowing down a potentially productive route through an interface. An example of a cost is the fact that undoing a selection of a top-level menu (by moving the mouse cursor away from the menu) is significantly cheaper than undoing the selection of a pulldown menu which often involves interacting with a dialog box. The use of prior knowledge has a cost in that the user may need to apply recall strategies, consider synonyms and related terms, and form analogies to experiences with other software.

Models can be represented by formalisms such as a production system. This system is composed of a number of rules (productions) which represent knowledge and each production consists of a condition component and an action component. The system performs the action if the condition is found to be true. These are called recognize-act cycles. Learning is generally represented in these systems through:

- *production composition* – collapsing a sequence of productions that are used to solve a problem into a single production that does the same thing

- *proceduralism* – the process of building domain-specific declarative knowledge directly in the productions so that there is no need to hold this knowledge in working memory [Anderson, 1983 in van Oostendorp, and Walbeehm, 1995]
- *strengthening* – each time a production rule is used its strength is increased and the time to execute a production is a function of its strength [Nilsen et al., 1993].

A problem with a number of models found in the literature is that the predictions for execution time and learning time are based on ideal behaviour. Van Oostendorp and Walbeehm [1995] propose research directions for the extension of current HCI modelling techniques to include exploratory learning in the context of direct manipulation interfaces. They propose a method to account for slips (errors) by allowing for partial matching of productions based on a weighting and a threshold mechanism. They also propose a method of modelling the fact that recognizing is easier than retrieving.

The previously mentioned model by Rieman, Young, and Howes [1996] and that proposed by van Oostendorp and Walbeehm [1995] are representative of the current state of cognitive modelling. These are based on many predecessors which include GOMS, CCT, EXPL, and CE+. The description of these models is beyond the scope of this paper, however, the interested reader is encouraged to look at "The Growth of Cognitive Modeling in Human-Computer Interaction Since GOMS" [Olson and Olson, 1990] and "Theory-Based Design for Easily Learned Interfaces" [Polson and Lewis, 1990] for further information.

Bösser [1987] compares modelling and prototyping as two different means to evaluating system design. Prototyping implies an empirical evaluation of the prototype which he asserts is lengthy and expensive if done properly. In contrast, he argues that modelling allows for fast evaluation of the formal properties of the model. Bösser suggests that it is ideal to employ both methods in combination such that the space of possible design alternatives is modelled and then reduced to a smaller number of designs, which can be evaluated in the form of prototypes.

Bösser [1987], in the last chapter of his book, describes a way to model learning requirements. Based on my reading of this section, the description wasn't nearly clear enough for me to implement his learning model.

3.5 Designing for Learnability

3.5.1 Designing for Exploration

Shneiderman [1997a] recommends a strategy that permits a *level-structured* (sometimes called *layered* or *spiral approach*[7]) to learning when diverse user classes must be accommodated by the system. He says that novices should be taught a minimal subset of

[7] This approach is based on the spiral model for software engineering [Boehm, 1988].

objects and actions with which to get started because they are most likely to make correct choices when they have only a few options and are protected from making mistakes. This is in essence a form of functionality blocking where the functionality is layered. Unfortunately, Shneiderman doesn't suggest specific design guidelines to accomplish this design strategy.

Carroll and Mack offer a number of general suggestions for exploratory environments [1984]:

1. Learners should be made to feel responsible and in control. An exploratory environment establishes and reinforces a role of responsibility and control for the learner via the system interface and training materials.
2. An important property of an exploratory environment is system simplicity.

Carroll et al. proposed the following for successful guided exploration of computer systems [1986, cited in van Oostendorp and Walbeehm, 1995]:

1. help users set the appropriate goals
2. offer helpful hints on how these goals might be achieved
3. provide users with checkpoints and means of confirming that they are heading in an appropriate direction

Norman's three requirements for an explorable system are [1988, cited in van Oostendorp and Walbeehm, 1995]:

1. All possible actions should be visible at all times and the user should be able to perform every single one of these possible actions.
2. The effect of every action should be visible and readily interpretable.
3. Actions should be without cost: whenever an action leads to an undesirable outcome, the user should be able to nullify it. For operations that are not nullifiable, both an appropriate warning and the opportunity to cancel the operation should be given.

Rieman, Young, and Howes provide a number of design implications related to generating labels in the interface [1996]:

1. Exploratory action will not be attempted until the user has balanced its predicted "safety" against the quality of its label. The safest actions are those that are trivial to undo or those that have a predictably minimal effect.
2. Choosing good labels is not always easy. Designers need to be aware of actions that may have multiple equally good labels. Some users may become blocked when they don't find the expected label and designers should provide assistance for these actions in the form of online materials and manuals.
3. Designers should supply additional information that could shift the balance at the point where benefit-to-cost comparisons are most difficult. When users pause with the cursor over a menu item (which occurs frequently in secondary passes of the menu) the system could recognize the pause and display the system state (dialog box or whatever) that would be evoked if the user selected the given action. At this point the user can select the item or can move the cursor which will remove the system state (dialog box or whatever) automatically without the user having to figure out how to get out of it.

3.5.2 Designing for Error

An important system property is safety. Safety is defined by Carroll and Mack [1984] as "the capacity of the system to protect the learner from demoralizing penalty." It is paramount that the learner must feel safe in taking action.

Designing for error is one way to enhance learning in general and more specifically exploratory learning. Norman [1990] notes that when an error in interaction occurs between the user and computer, the initial reaction is usually to attribute this error to the user when in fact we should question the design of the system instead. The work required by the user to learn a system is significantly reduced when a system is designed to minimize interaction error and to provide appropriate error messages when an error state is reached. Lewis and Norman [1986] differentiate two types of errors: mistakes and slips. A mistake occurs when the user has the wrong intention to begin with. A slip occurs when the user has the correct intention but performs an action that was not intended. They say that error can be minimized by using the appropriate representation (e.g., using point and click on an icon to open a file can minimize "file not found" type of errors) and by avoiding false understandings.

One of the first hurdles with error recovery is the detection of the error itself. Slips are easier to detect than mistakes because the outcome of the action in a slip is different than the intention. With a mistake it is the intention that is wrong to begin with. Six possible ways in which the system can respond to help the user detect the error are [Lewis and Norman, 1986]:

1. **Gag**: Gag is a forcing function. This is something that prevents the behaviour from continuing until the problem has been corrected.
2. **Warn**: Warning is less obtrusive than gag. Warning basically tells the user there is a problem but allows the user to continue despite the warning.
3. **Do nothing**: Do nothing basically means that no system response is given if the user attempts an illegal action.
4. **Self correct**: When the system detects an illegal action it can try to correct the action. An example of this given is simple spelling correction. Systems that use self correction must have undo features in case the corrected action is not the intended action.
5. **Let's talk about it**: This method occurs when the system engages the user in a form of dialog when a problem is detected.
6. **Teach me**: This method occurs when the system queries the user on the intention behind an action and learns from the user.

Although these methods help in the detection of a problem, they don't necessarily help identify the problem.

3.5.3 Learner-Centered Design

Soloway, Guzdial and Hay [1994] have argued that the HCI community must move beyond "user centered" design to "learner-centered" design. The goal of design should be to support individuals' development of expertise and the development of deeper understandings of content and practices. Despite the fact that this goal is broad and encompasses the spectrum of learners, the learner-centered movement has primarily focussed on students in classrooms who are learning content. This is not to say that researchers in the area of learner-centered design do not recognize that learning is also for professionals. It is simply that the research to date has not addressed this group of learners.

The learner-centered design movement was begun by Elliot Soloway at the University of Michigan. Soloway explicitly states that the current focus for him and his students is on K-16[8] learners but he does cite Senge's [1990, cited in Soloway et al., 1996] compelling arguments that an organization must be a learning organization in order to be productive and thus concludes that learner-centered design should also have validity in the workplace [Soloway et al., 1996].

According to Soloway, Guzdial and Hay [1994], the following special needs of learners must be addressed when putting learners at the center of the design:

- **Understanding is the Goal:** How can learners acquire domain knowledge from an application? For example, how do users come to know accounting principles and practices when a spreadsheet is presented to them?
- **Motivation is the Basis:** Learners tend to procrastinate when confronted with a task for which they are unprepared. How can software play a role in supporting the learner's wavering motivation?
- **Diversity is the Norm:** How can a single application support learners who are from a diverse set of backgrounds, with a diverse set of interests, skills and abilities?
- **Growth is the Challenge:** An application is by and large the same on day 1 as it is on day 100. But a user can be very different. For example, that person may have learned quite a bit about a problem domain and might have developed a set of skills and practices in that domain. How can the software accommodate the change in the user?

The term *scaffolding* is used frequently in learner-centered design research. Scaffolding predates the use of computers in the classroom and in general is a technique for providing support to learners while they are learning a new task [Wood, Bruner, Ross, 1975 and Rogoff, 1990 cited in Soloway, Guzdial and Hay, 1994]. For example, the provision of scaffolding through human tutors has been well established as an effective means of supporting learning [Jackson, Krajcik, and Soloway, 1998]. Scaffolding essentially allows a learner to engage in activities that would otherwise be beyond his/her abilities. As the learner develops the required knowledge and skills, the scaffolding fades so that the learner is fully in control.

---

[8] Soloway does not clarify what he means by K-16. He is likely referring to the school years K-12 plus four years of undergraduate education.

Soloway, Guzdial and Hay [1994] propose the TILT Model (Tools, Interfaces, Learner's needs, Tasks) to guide the design of learner-centered software. The objective of the model is to highlight how software might address the special needs of the learner. The TILT Model uses specific scaffolding strategies that are appropriate for the needs of the learner. For *Tasks,* a coaching scaffolding technique is recommended to help students acquire knowledge and the specific practices of a task domain. For *Tools,* scaffolding is provided by making the tools adaptable such that they support a learner growing in expertise. For the *Interfaces* to the tools, scaffolding can be provided through the use of different media and modes of expression.

Jackson, Krajcik, and Soloway [1998] use the terms "fadeable supports" to describe scaffolding. They argue that although many techniques have been explored that provide various supportive structures for learners, typically the support does not fade within the software itself. But the scaffolding must fade as the user develops expertise in the same way that a human tutor provides less and less support as the tutee acquires knowledge. They list *adaptive* and *adaptable* interfaces as potential solutions. Theoretically, adaptive interfaces change automatically using a model of the learner's understanding. In practice, however, an extensive model of the learner's knowledge may be hard to specify or evaluate in more open-ended domains. Adaptable interfaces, on the other hand, put the user in control of the fading. Because it may be hard for the learner to make fading decisions, the software can be designed to help the student measure his or her progress and understanding. Jackson, Krajcik, and Soloway have developed a design approach called Guided Learner-Adaptable Scaffolding which "is designed as discrete, fine-grained scaffolding of various types, faded under control of the learner, with guidance from the software to aid the learner in making informed decisions."

Jackson, Krajcik, and Soloway [1998] identify three categories of scaffolding: *supportive*, *reflective*, and *intrinsic*.

Supportive scaffolding provides support for doing the task. The task itself is unchanged and so as supportive scaffolding fades, the task is the same as before, however, it is expected that the learner has internalized the concepts which have been scaffolded. This kind of scaffolding includes guiding, coaching, and modelling and is the most often referred to as scaffolding in the literature [Jackson, Krajcik, and Soloway, 1998]. Guiding scaffolding can be provided through messages which appear when appropriate and which can be faded through a "stop reminding me" button. Coaching and modelling scaffolds can be provided through contextualized help buttons. These buttons may not fade per se, but are faded simply through not being invoked.

Reflective scaffolding is support for thinking about the task (e.g., planning, making predictions, evaluating). Like supportive scaffolding it doesn't change the task itself, but instead makes the activity of reflection explicit by eliciting the learner's thoughts. It can be provided by a notepad that appears alongside the application's main window where the learner is encouraged to reflect by typing plans, descriptions, predictions and evaluations.

Jackson, Krajcik, and Soloway [1998] use the term intrinsic scaffolding to mean support that changes the task itself, through reducing the complexity of the task and focusing the learner's attention and by providing mechanisms for visualizing or thinking about a concept. Ideally scaffolding should support gradual fading such that the task is gradually changed but associations remain which enable the learner to progress to more complex and abstract tasks. Intrinsic scaffolding can be implemented as defaults which hide all but the simplest tools from the novice learner, but make advanced features available as the learner grows and develops expertise. Intrinsic scaffolding is manifest as different views and representations of model components, or different sets of enabled controls and tools.

Norman and Spohrer provide an introductory section to a 1996 issue of the *Communications of the ACM* (volume 39, number 4) which focuses on learner-centered design. They evaluate the papers on learner-centered design from this issue along three dimensions: engagement, effectiveness, and viability. Engagement determines motivation. Motivation correlates well with time on task and they note that it can make more of a difference between success and failure than any other factor. Effectiveness is concerned with the actual learning that takes place. Viability is concerned with the feasibility of creating learner-centered software. Norman and Spohrer note that the primary strength of the articles presented in this issue is that of engagement. The dimensions of effectiveness and viability were not the focus of the articles and so they are limited in these areas. The assessment of effectiveness is limited to the opinions of students and teachers which is not robust. However, the authors note that conventional assessment that is based on a rigidly controlled question and answer format is probably not the correct solution. In terms of viability, they note that this is the most difficult dimension to assess and that to address this issue would require the complete development of a curricula and deployment in school systems.

A number of applications have been implemented in learner-centered design research. Some examples that are drawn from the aforementioned issue of *CACM* include: Broadcast News, a multimedia tool that teaches social studies to high school students by allowing them to determine the content of a television news story [Schank and Kass, 1996]; Cardiac Tutor, a knowledge-based simulation for teaching about cardiac resuscitation and the Engineering Tutor which teaches the concepts of design for manufacturing, specifically design for injection molding, to first-year engineering undergraduate students [Woolf, 1996]; scaffolded examples for learning object-oriented design which are realistically-sized sample problems whose complexity is gradually revealed in steps that leverage and reinforce the intrinsic structure of the problem-solution process [Rosson and Carroll, 1996]; and Model-It, a modelling tool that enables students to gain insight into the behaviour of complex systems [Jackson et al., 1996].

3.5.4 Questioning the All-in-One Model

The worth of complex all-in-one tools is being questioned [Sagar, Hof, and Judge, 1996; Buxton 1998]. Buxton suggests that too much functionality is often overloaded into a single device. The result is that the device is a general device that can do most things but nothing really well and so it can be considered "weak". Further, the cognitive load associated with using the device is proportional to the number of features associated with the device. It follows that the complexity and thus the cognitive load associated with using a complex tool can be reduced by making the tool less general. Reducing the cognitive load implies ease of learning.

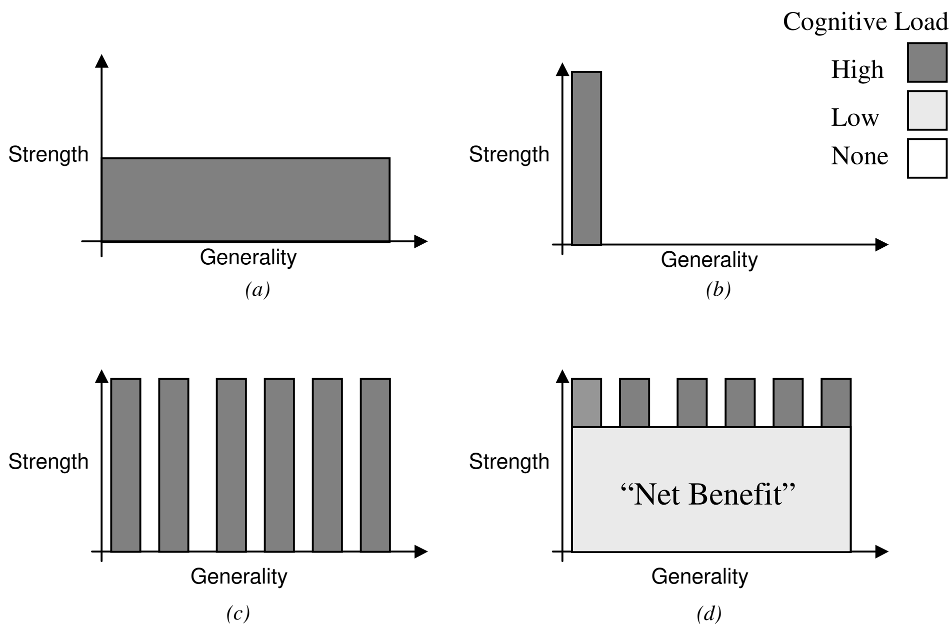

Figure 3: Representation of a (a) General-Weak tool (b) a Strong-Specific tool (c) a Strong-Specific toolset and (d) networked Strong Specific toolset (slightly adapted from Buxton [1998]).

The rectangular area in these graphs represents a single tool (e.g., word processor) and the area within the rectangle represents the cognitive load associated with using this tool. The x-axis represents the degree to which the tool is a general-purpose tool. The y-axis represents the degree to which the tool will allow you to accomplish specific tasks.

The idea is that instead of designing general tools that incorporate enormous amounts of functionality but don't have significant strength in many of the functionality domains, it is better to design individual tools that have very specific purposes. Not only will the strength of the individual tools be greater but the cognitive load will be lessened because the user can select which tools to use.

If one only needs specific functionality, then it is better to use a specific tool instead of a general tool because the cognitive load required is less. This can be seen by comparing the area under the curve in Figure 3(a) versusFigure 3 (b). If one only needs some of a general tool's functionality, one might be better off selecting a few specific purpose tools. The load associated with two or three tools shown inFigure 3 (c) may be less than that inFigure 3 (a). But what if all the functionality is needed? Clearly the load associated with all the tools inFigure 3 (c) is greater than that inFigure 3 (a). For this case Buxton[1998] advocates the model shown inFigure 3 (d). Here, the tools are integrated intelligently, removing most of the cognitive load from the user. The example he gives is that of using a phone while driving a car and listening to the car stereo. If these tools are unaware of each other then the cognitive load would be high. Imagine a situation in which the stereo is playing loudly and the phone rings. In order to answer the phone the user must not only fiddle with the phone to answer it but must also adjust the stereo. If, on the other hand, these tools are integrated, then when the phone rings, the stereo would know to turn itself down. Buxton refers to this as a net benefit.

4 Intelligent User Interfaces

Research in the area of intelligent user interfaces provides some insight into the issue of designing for learnability in heavily-featured systems. Although there is no clear definition of what constitutes an intelligent user interface (IUI) [Encarnação, 1997], at the most abstract level one might want to characterize such interfaces as ones that attempt to bridge the design/learning gap through the application of intelligence. Interfaces need not be static, the same for all, and void of any dynamic understanding of what the user is trying to accomplish. Rather, interfaces can dynamically reconfigure to accommodate individual differences, or can do mundane tasks that the user would rather not do, or perhaps even have a "richer" dialog with the user than what is provided by point-and-click.

Research in IUIs has been underway for the last three decades. Miller, Sullivan, and Tyler [1991] provide a brief background on this research. They note that some of the early research in Artificial Intelligence (AI), including natural language and problem solving research, falls within intelligent interface research. The goal of this AI work was to address how people might interact with systems capable of solving large, complex problems. This early work was dominated by the metaphor of natural language-like discourse where the user asked questions and the computer replied. They note that the vision shifted in the 1980s largely because of the introduction of the graphical user interface and also because natural-language understanding remained such a challenging area of research. It was felt that GUIs could make it easier for intelligent systems to determine the meaning underlying users' actions: "instead of having to search for the meaning in a natural-language statement, a graphical interface can be built around the important concepts in the task and domain at hand, making the content of a user's actions immediately accessible to an underlying reasoning system." Despite the latter, Miller, Sullivan, and Tyler are clearly not saying that direct manipulation and GUIs are always better than natural language or other agent-oriented interfaces. In fact, they believe that what is needed is a synthesis of the two perspectives.

Mark [1991] provides the essence of more recent IUI research in the forward to Sullivan and Tyler's [1991] book. He notes that the premise behind IUI research is that software applications have become significantly complex and that despite advances in interface design techniques, it is difficult to design an interface environment that allows users to act intelligently:

> "In a complex environment, it is very difficult to distill out a set of interface controls that deliver the application's power, but that human beings can learn and remember; it is very difficult to define interface behaviour that treats users fairly - does not hector them, mislead them, punish them for experimentation, and so on." [p. vii, Mark, 1991]

The approach adopted by IUIs is that the interface is in effect promoted to a team member:

> "The goal is no longer to design a user interface that has a complete set of usable controls and understandable behaviour. The new challenge is to give the interface some understanding about what the users are trying to do and how they need to go about doing it. ... The interface must be given enough knowledge of the problem to allow it to take on significant responsibility. The interface must be able to communicate with the user at the level of taking direction and giving advice about tasks - if the user has to communicate at the level of detail within tasks, no significant delegation has taken place. The interface must be able to explain its activities in order to allow the user to build up confidence that it is to be relied upon." [p. viii, Mark, 1991]

Mark [1991] acknowledges that whether all of this is in fact *intelligence* is a matter of expectation and perception.

Encarnação [1997] addresses the shifting definition of intelligent interfaces. He notes that at one time the definition of intelligent user interfaces included "...the integration of an AUI (adaptive user interface)...both with an intelligent help system (IHS), making context-sensitive and active help available, and with an intelligent tutoring system (ITS), supporting the user in learning the use of the system," where as in a common definition today, the term 'intelligent' refers to any of these realizations, but not necessarily their entirety. Dieterich et al. noted in 1993 that the difference between AUIs and IUI was not well defined.

Based on the review of the literature, it is still not clear where the boundary between AUIs and IUIs lies. The earlier literature seems to use *adaptive interface* in place of or interchangeably with *intelligent interface*. Recent classifications such as that by Encarnação [1997] only serve to confuse the issue further. In a figure he displays AUIs as a superset of IUIs but has "Interface Adaptability" as a component/subset of IUIs. For the purposes of the current review, the broader research area will be referred to as IUIs, and AUI will be reserved to mean interfaces in which the functionality accessible to the user is adapted based on user characteristics.

The literature covering intelligent user interfaces presents a number of recurring themes and therefore it is possible to impose a rough categorization. The most general categories include *agents, adaptivity, user modelling, task modelling/plan recognition* and *multimodal communication*. These should not be seen as mutually exclusive areas of research by any means. *Intelligent help, intelligent tutoring*, and *dynamic multimedia presentation* are categories of applications that are based on the previously mentioned broader categories. The *architecture* and *evaluation* of intelligent systems is also covered in the literature.

4.1 Agents

There has been great controversy over agents, the foremost contention perhaps being the lack of a clear definition. The debate over using agent technology in the interface versus more established techniques such as direct manipulation has been heated. Highlighting the issues raised in this debate serves as a reasonable introduction to agents. In two recent debates between Pattie Maes, a leading researcher in the area of agent technologies, and Ben Shneiderman, a long time proponent of direct manipulation, it seems that some form of consensus may be on the horizon [Shneiderman and Maes, 1997]. Maes acknowledged that the word *agent* is overloaded. The area of agent research that relates to complex technology is what she calls *software agents.* She specifically prefers avoiding the terms *intelligent agents* and *autonomous agents* because they are problematic. Maes gives the following advantages of software agents:

1. A software agent knows the individual user's habits, preferences, and interests.
2. A software agent is proactive. It can take initiative because it knows what the user's interests are. It can, for example, tell the user about something that he/she may want to know about based on the fact that he/she has particular interests. Current software is not at all proactive. All of the initiative has to come from the user.
3. Software agents are long-lived. They keep running, and they can run autonomously while the user goes about and does other things.
4. Software agents are adaptive in that they track the user's interests as they change over time.

An agent can act on the user's behalf while he/she is doing other things in much the same way that a travel agent will act on an individual's behalf once his/her travel needs and preferences are made known. An example of an agent is presented by Kozierok and Maes [1993] in which the agent learns the user's preferences and needs with respect to scheduling meetings by observing the user, through reinforcement (direct feedback from the user), and by direct instructions from the user.

Maes notes the following reasons that we need software agents today:

- Our computing environment is no longer closed and under a user's complete control like it once was. Our computer provides a viewport into a vast and dynamic network of information and other people.
- The typical user is no longer a computer professional.
- People use their computer for more and more tasks and are thus required to keep track of more and more information.

Shneiderman [1995] has argued in the past that "the effective paradigm for now and the future is comprehensible, predictable, and controllable interfaces that give users the sense of power, mastery, control, and accomplishment" and that the term agent itself is ill-defined but seems to include the following components:

- anthropomorphic presentation
- adaptive behaviour
- accepts vague goal specification
- gives you just what you need
- works while you don't
- works where you aren't

He notes that the first three seem appealing at first, but have proven to be counterproductive. The latter three are good ideas but can be achieved more effectively with other interface mechanisms.

Despite some of Maes earlier research directions with agents, she states that it is a misconception to think that agents are necessarily personified or anthropomorphized. In fact, she notes that most agents are not. Maes also clarifies that agents do not necessarily rely on traditional AI techniques, like knowledge representation and inferencing. Many of the commercially available and successful agents rely on either user programming or machine learning rather than traditional AI techniques.

Maes argues that agents are not an alternative for direct manipulation, but rather they are complementary metaphors: an agent is not a substitute for a good interface. She argues that the reason for agents is delegation and that no matter how good the interface, there are some tasks that she just may not want to do herself. She gives the example that if her car had a perfect interface for fixing the engine, she still would not fix it.

Maes also addresses the criticism that agents make the user dumb and that they usurp all control from the user. Concerns about agents are addressed by the guidelines for agents presented by Maes at the 1997 International Conference on Intelligent User Interfaces [Computer Science and Telecommunications Board, 1997]:

- Make the user model available (inspectable, modifiable) to the user.
- The agent’s method of operation should be understandable to the user.
- The agent should be able to explain its behavior to the user.
- The agent should have the ability to give continuous feedback to the user about its state, actions, and learning.
- The agent should allow variable degrees of autonomy, and the user should decide how much and what type of tasks to delegate to the agent. The user should be able to “program” the agent (e.g., teach it things, make it forget things).
- The user should not have to learn a new language to deal with the agent. The goal is to use the application to communicate between the agent and the user.

These guidelines serve to move the agent research much closer to Shneiderman’s requirements for controllable and predictable user interfaces. Further, Maes noted at the second of the two debates that one of the key reasons for the division in philosophies is that the two camps are focusing on different problem domains. The Shneiderman camp is dealing with a well-structured task domain and a well-organized information domain, so that it lends itself to visualizing all of the different dimensions: for example, visualizing the information stored in a database. The Maes camp, on the other hand, is dealing with an information domain that may be very ill structured and very dynamic; an example of such a domain is the World Wide Web.

Shneiderman [1995] cites a number of examples where anthropomorphic terms and concepts have continually been rejected by consumers and, in fact, Maes notes that the most successful software agents thus far are ones that are pretty much invisible. Despite this, there are at least a few examples where agents are visible and are used to help the user navigate the interface itself. These agents do not adhere to the delegation philosophy of agents and demonstrate that research on agents is diverse. There is the Microsoft Office personal assistant which is initially instantiated as an anthropomorphic paper clip to whom the user may ask natural language questions and who will also suggest somewhat unobtrusively more efficient techniques for a user's current task. At this point I haven't seen any literature that covers the user's response to this agent. A second example by Rich and Sidner [1996] is called a collaborative interface agent. This agent mimics the relationships that exist when two humans collaborate on a task involving a shared artifact such as two mechanics working on a car engine together or two computer users working on a spreadsheet together. There was no user testing documented on this research.

Dryer [1997] provides a brief discussion of *wizards* and *guides* which are perhaps the most common kind of UI agent today. Wizards are most common. Their goal is to assist the user by breaking a complex task into a series of steps and then present one step at a time to the user. Wizards work best with a linear series of steps and are therefore most successful when the tasks have algorithmically derived solutions. Dryer notes that these agents generally do not use any artificial intelligence although they are sometimes perceived by users to be intelligent. The benefits of a well-designed wizard are that a multi-purpose task interface is replaced by a task specific interface that guides the user along an efficient path to task completion and that autonomously completes those steps of the task that do not require the user's attention. One possible disadvantage of such a constrained process is that the user may not actually reflect on his/her actions and therefore may not learn from the process.

Based on Dryer's [1997] account, it wasn't entirely clear what guides are. He provides the following description: "Guides are another kind of UI agent. Typically, guides provide task assistance by monitoring a person's interaction with the information system and presenting information appropriately. ... Guides are intelligent because they annotate an interface whenever and however it is most likely to be useful. A well designed guide will direct a person through the next step in a task."

According to Dryer [1997] guides are best for frequent tasks because users want to learn about tasks they do frequently. Wizards are best for infrequent tasks that users don't necessarily want to learn about but do need to accomplish. Wizards work best for solutions that can be derived algorithmically whereas guides can assist either algorithmic tasks or heuristic tasks.

Myers et al. [1993] discuss the use of heuristics[9] to predict users' intentions. They note that systems that use heuristics attempt to delegate some of the low level details in order to save the user time and are also hopefully easier to learn because the system does part of the work. The disadvantages are similar to those already mentioned: that an incorrect action might occur which might not be noticed by the user; the user might not understand why the system did a different action or how to get the desired action to occur; again that the user might have the feeling that the system was unpredictable and that he/she no longer had control; the additional user testing costs to determine whether the heuristics are sufficiently predictable to users and to tune the heuristic or its presentation; and the additional documentation and quality assurance costs. Myers et al. suggest that successful use of a heuristic requires that:

1. A majority of users would predict the same result of an action performed in a given context.
2. An algorithm can be developed which interprets the context and produces the result most users expect.
3. In cases where the algorithm does not do what a user requires, it should still give a result that is interpreted as reasonable, and the result must not be harmful.
4. It should be undoable.
5. The user can discover ways to override the default behaviour when necessary.

These guidelines for heuristic use and Maes' guidelines for agents show the importance of user expectation and control when some tasks are delegated to the system.

4.2 Adaptive User Interfaces

Research in adaptive user interfaces (AUI) addresses the diversity of the user population differently than agent research. Rather than delegating cumbersome tasks to a trustworthy agent or collaborating with an agent in order to navigate the interface, the philosophy of AUI research is that the interface itself should adapt to the needs, preferences, and skills of the user. The goal is for the complexity of an adapted interface to be less than that of an equivalent all-in-one interface because the functionality accessible matches the user's needs, preferences, and skills. Cote-Muñoz [1993] noted the following consequences of AUIs: the user will better master the complexity of software, there will be better user performance, the system will be able to gain and maintain the attention of the user, and the system will avoid underloading or overloading the user.

The question of who adapts the interface is relevant. Tyler and Treu [1989] identify and discuss three possible sources of adaptation: a computer expert, the user, or the system. In the first case, a computer expert, perhaps the original designer of the system, modifies the

[9] 'Heuristics' are a problem solving technique in which the most appropriate solution is chosen using rules [Myers et al., 1993].

interface based on user feedback. The problem with this scenario is that it doesn't scale; an expert may be able to construct a few differently-tailored interfaces, however, this will not likely be sufficient to meet the needs of all the different users. Another problem with this that was not mentioned by Tyler and Treu is the substantial delay or turn around time generally required for modifications. The second possibility is for the user to take advantage of the customizability that most systems provide by tailoring the interface to suit his/her own style and abilities. This scenario works well for experienced users who know both what needs to be tailored and how to go about tailoring. However, it doesn't hold for inexperienced users who will not likely know what they need to tailor and how they should go about tailoring [Innocent, 1982; Page et al., 1996]. The last approach mentioned is that the system adjusts the features of the interface based on acquired knowledge about the individual user. The basic idea is that the system monitors the user and adjusts the interface to suit that individual.

Given that the goal of adaptive interface research is to match the interface to an individual's profile, the solution of having a computer expert in charge of adapting the interface is not practical. This leaves the adaptation up to the user or the system itself. Dieterich et al. [1993] provide a survey of the AUI literature and a framework for understanding the adaptation process. They identify four stages in the process of adapting a user interface:

1. initiative: the need for adaptation is suggested
2. proposal: alternatives for the adaptation are proposed
3. decision: one of the alternatives is chosen
4. execution: the chosen alternative is executed

At each of the four stages it is either the user or the system that is in control. For example, when the system controls each stage, the system is essentially a self-adaptive system. When the user is in control of all stages, the user is doing the adaptation which is tantamount to tailoring or customization. (These two extremes are sometimes juxtaposed as *adaptive* vs. *adaptable* systems [Fischer, 1993].) Then of course there are different combinations. Figure 4(a) represents the configuration in which the software is entirely self-adaptive: the system is completely in control. Dieterich et al. [1993] note that the majority of research into AUIs has been from the system-adaptive perspective and they make the following conclusions from their survey:

- Systems that have user control seem more promising than those that have all/mostly system control. A configuration in which the user and the system share control, which they call Computer-Aided Adaptation, is deemed by the authors to be the most promising approach in order to obtain a user interface that will help the user to perform his tasks in a pleasant and effective way. This configuration is depicted in Figure 4(b).
- More effort should be spent on the integration of developed adaptivity mechanisms into common user interface design and management tools.
- Aspects of user acceptance and the evaluation of adaptation have been neglected in the past and need far more attention.

(a) (b)

Figure 4: Categorization of adaptive systems: (a) Self-Adaptive (b) Computer Aided Adaptation

The first point given above that advocates user control can be seen to parallel the more recent direction in agent research which acknowledges the importance of user control and user understanding of the agent behaviours.

To achieve the Computer-Aided Adaptation configuration, Kühme [1993] proposes an inspectable user model which gives the user an insight into adaptation strategies and underlying assumptions. The user should clearly be made aware of the existence of the user model and should have access to the included information. By changing the information in the user model, the user is essentially adapting the interface in an implicit manner. The user should also be able to adapt the interface explicitly by inspecting and adjusting the adaptation-related mechanisms. These include an adaptable dialog monitor which collects information relevant for adaptation and an adapter which adapts the dialog by applying the information represented in the user model.

Greenberg and Witten [1985] conducted some early work on adaptive interfaces and noted some advantages and disadvantages to adaptation. The advantages are:

1. variations in expertise across users
2. evolving user needs
3. user has appropriate control
4. attempts to rectify user-designer conflicts

The disadvantages are:

1. dynamics of user-system concurrent modelling (at the same time the adaptive system is trying to make a model of the user, the user is trying to model the system)
2. user will lack confidence in a system that seems inconsistent (although Grudin [1989] argues successfully that consistency is only one of many competing design goals)
3. user does not have appropriate control
4. complexity of implementation
5. inaccuracies of model construction
6. difficulties in evaluating adaptive systems

From the above we can see that control, which appears both as an advantage and a disadvantage, is indeed controversial and further it is not clear what constitutes appropriate control.

Stephanidis, Karagiannidis, and Koumpis [1997] document a methodological approach to adaptive systems. They note that in most adaptive systems, the adaptation strategy is hard-coded into the system and therefore when changes are needed, it is relatively difficult to implement the changes. They focus on the adaptation strategy as a decision making process, which is characterized by the following attributes:

- *what to adapt*: aspects of the user-computer interface that are subject to adaptations are called adaptation constituents and can be semantic, syntactic, or lexical;
- *when to adapt*: aspects of the interaction called adaptation determinants on which the adaptation decisions are made;
- *why to adapt*: the adaptation goals underlying the adaptation process;
- *how to adapt*: adaptations are driven by a set of rules, adaptation rules, that essentially assign certain adaptation constituents to specific adaptation determinants for given adaptation goals.

Stephanidis, Karagiannidis, and Koumpis [1997] describe a methodological approach that enables:

- the customization of the set of adaptation determinants and constituents;
- the incorporation of the adaptation goals as an integral part of the adaptivity process;
- and the modification of adaptation rules, according to the goals of adaptivity.

4.2.1 Examples of Adaptive User Interfaces:

Adaptive Prompting

Malinowski et al. [1993] introduce the idea of adaptive prompting. This technique aims to reduce the user's confusion caused by the volume of prompts (menu items, dialog boxes, etc.) by leading the user to the functionality which is most relevant in a given situation. Adaptive prompting is provided to the user through a complementary preselection of the relevant options and is not a substitute for existing interaction techniques.

The authors discuss two different adaptive prompters. The first is the *Adaptive Action Prompter* which is a permanently visible, dynamic menu (or control panel) which will include only the most appropriate and most likely to be chosen actions based on the user's context. Thus the prompter contents are updated with every context change. The user can always select actions from either the prompter or the regular menus, whatever is more convenient in a given situation. Because the prompter lists the most appropriate actions in one place the user has a good survey of sensible alternatives. The prompter can be shown as plain menu items alone, or menu items that indicate the referred object (e.g., "CUT selected text" where selected text is the referred object), or menu items with task-oriented explanation (e.g., "START to start the simulation using the selected sample"). The user can optionally be involved in controlling the rules used for the prompting.

The second form of adaptive prompting discussed is *Adaptive Dialog Boxes*. This prompting is to address the problem that dialog boxes often present a lot of parameters that are required for a function. The authors suggest that strategies for coping with the volume of parameters such as sorting the parameters based on frequency of use or moving rarely needed parameters to an additional dialog box titled something like *More Parameters*[10] change the layout of the dialog box and therefore results in confusion to the user. The strategy of adaptive prompting in dialog boxes is to present the information in a way that allows the user to identify the important items and their parameter settings at a glance. This is achieved by using forms of highlighting and color coding to draw the focus of the user. The structure of the dialog box is not changed in this approach. No user testing for either prompters is documented.

AIDA

Cote-Muñoz [1993] documents an adaptive system for interactive drafting and CAD applications that fixes the amount of functionality offered by icons and menus based on the user's knowledge. But to support exploratory learning, the user has access to the full system's functionality through a command line. In addition, new functionality is introduced in one of two ways; either the user creates a new command/macro or the system recognizes a repetitive operation (such as creating three lines that result in a triangle) and suggests the creation of a command/macro. These command/macros become available to the user through new menu items. No user testing is documented.

SAI - Skill Adaptive Interface

According to conventional wisdom, direct manipulation interfaces are better for novices and command line interfaces are better for experts. Some attempts to create hybrid applications (applications that include both direct manipulation and command line interfaces) have been made. Gong and Salvendy [1995] studied hybrid systems and found that most users never moved beyond using the menu. To rectify this they created the Skill Adaptive Interface (SAI) that gently pushed the user to use the command line. Once a user had selected a menu item a threshold number of times, all subsequent times the same item was selected the system would provide a prompt which gave the equivalent command. The user was then forced to use the command line for this item. Gong and Salvendy reported user studies that showed this hybrid technique to have promise.

In addition to gently forcing a user into command line usage, SAI was also adaptive in that the menu contents were variable. Once the user was able to enter the command at the command line without using the prompt available from the menu, the menu item was transferred from the active menu to a hidden menu that was attached at the end of the active menu. Users had access to the hidden menu, although the authors do not specifically say how. The hidden menu contained items for which the users had already learned the

[10] This solution of having a button in a dialog box that permits access to more advanced options appeared in the Xerox Star. The term "Progressive Disclosure" has been used by the original Star design team to describe this model [Johnson et al., 1989].

corresponding command and also items that the user had not yet encountered. When an active menu item was transferred to the hidden menu, an empty menu slot was then available. To fill the empty slot, the system selected an item from the hidden menu that had not yet been made visible. The determination of which item should be selected for the empty slot was based on a priority parameter. The authors do not elaborate any further on how this parameter was set. This adaptive aspect of the system was not subject to user testing.

Adaptive Version of Microsoft Excel - Flexcel

Thomas and Krogsœter [1993] note that the great majority of research in adaptive systems has been done with prototype systems. One of their goals was to assess adaptation in a complex software product that was commercially used. They chose Microsoft Excel. They were able to modify the Excel interface through the Excel dialog editor and the macro programming language.

The following adaptation features were implemented:

- The user could define new menu entries and new key shortcuts for function parameterization.
- The user could define key shortcuts for Excel functions which normally can only be invoked from the menu.
- For functions with default parameters, the default could be changed.
- A separate adaptation toolbar was made available to make the aforementioned tools more visible.
- System generated adaptation suggestions were indicated by an acoustic signal and a blinking button. The user could access the suggestions at his/her convenience. Unread suggestions were maintained in a tip list.
- Usage suggestions reminding the user of seemingly forgotten adaptations, and a critique model telling the user how the adaptation tools may be used more efficiently were also included.

In general, the system was well received in user testing although the paper didn't specify some key details including: the number of users tested, the duration of system use, and the background knowledge of the users.

UIDE - The User Interface Development Environment

The core of UIDE is a knowledge base [Sukaviriya and Foley, 1993]. An application is described in the knowledge base in such a way that no particular interface style is adopted - it is essentially just the functionality that is described in terms of instances of application actions. This is the domain dependent part of the knowledge base. Interface actions and interaction techniques reside in a domain-independent part. These interface actions can be linked to specific application actions and parameters which in essence specifies the interactions associated with each application function. More than one interaction technique can be linked to an interface action for alternative interactions.

Two types of interface adaptation are discussed: redesign of menus and dialog boxes, and addition of new commands (macros). They do not provide enough detail to determine exactly what is meant by “redesign of menus and dialog boxes” but one can assume that it has to do with altering the contents of the menus and dialog boxes by re-ordering the contents or adding and deleting items.

Both the temporary loss of productivity due to relearning and the longer-term productivity gain due to the reorganization are factors considered by the system in determining whether adaptation should occur. To suggest command macros, the chronological history of interactions is periodically examined to find repeated sequences of commands. No user testing is documented.

4.3 User Modelling

In order for a system to self-adapt or to suggest adaptations that are appropriate to the user, the system must have knowledge about the user. This knowledge is generally maintained in a knowledge base called a user model, which is responsible for acquiring and managing data as well as providing means for the application (or consumer of the user model) to access the data. This user model should not be confused with what are sometimes referred to by the same name in HCI research. User models in HCI may refer to the user’s mental model or the designer’s model of the user [Norman, 1986].

Kay [1993] defines a user model as “a collection of information that constitutes a model of the user. This information is explicit and it is a separate entity...It is emphasized that this model, like all models, attempts to represent only some aspects of the user, namely those that are relevant to the domain at hand.” Greenberg and Witten [1985] provide the perspective that a user model is the computer’s model of the user.

There are a three main issues in user modelling: what user information is relevant, when and how the information will be acquired, and how it will be represented in a user model.

what user information is acquired?

Encarnação [1997] provides the following list of the types of user information that are generally of interest in user modelling:

1. existing or missing knowledge of the user
2. goals and plans of the user
3. user preferences and tendencies
4. user experiences and skills
5. user misunderstandings

how and when is the user information acquired?

Kühme [1993] notes that there are two sources of relevant information about the user. The first is an explicit method in which the user provides self-estimations and preferences through question-and-answer type sessions at isolated times during the interaction. The second method is implicit where the system attempts to deduce information by constantly monitoring the user's dialog with an application [e.g., Vaubel and Gettys, 1990]. Kühme notes that there are problems with both self-estimations and deduction through dialogue monitoring. Self-estimations are not always reliable and deduction is "most often severely restricted by a very small user-system communication bandwidth". Encarnação [1997] has used the terms *separated acquisition* and *integrated acquisition* for these two methods. Further, he says that separated acquisition can be system controlled or user controlled. If the system controls when the user must complete the acquisition sessions, it is much more obtrusive then when the user is free to do it at his/her convenience.

how is the acquired user information translated into a user model?

Encarnação [1997] describes five different techniques used to create and represent a user model. The three most commonly used techniques are given below:

1. *Primary acquisition heuristics* use rules to build the user model on the basis of interaction with the user. These rules are generally dependent on the current application domain but there has also been work done on domain-independent acquisition heuristics.
2. *Stereotypes* are a natural way to generate initial or default values in a model [Kay, 1993]. When stereotypes are used, it is only necessary to gain sufficient information about the user in order to determine to which stereotype he/she belongs. Stereotypes are defined by the system designer and the number of stereotypes needed is often dependent on the application domain. It can be a significant disadvantage if a large number of stereotypes need to be generated [Kass and Finin, 1991]. The user can be associated with one or more stereotype.
3. *Overlay models* essentially represent an individual's knowledge as an overlay of the domain or expert knowledge. For each concept in the domain the user model contains an estimate of the user's knowledge about the concept.

Kay [1993] identifies some pragmatic issues that should be considered by the IUI research community in order for user modelling to move beyond prototype systems:

- need for tools that do the tasks involved in building and maintaining the user model;
- need to reuse models - models are expensive to create therefore the costs need to be amortized;
- need to address the issue of granularity - probably need to move away from heavy-weight and computationally expensive modelling to light-weight models;
- need to make the model accessible to the user - there is evidence that users want access and want to understand. Advantageous side effects are that the user can play an active role in constructing and verifying their model and the accountability of the programmer may be improved.

The development of user modelling shell systems have to some extent addressed the issue of model reuse and cost. The idea with these systems is that the system designer need not code all the necessary user modelling modules from scratch and embed them within the system but rather use a pre-coded modelling system that can be simultaneously active for other applications as well.

4.3.1 Examples of User Models:

UIDE – The User Interface Development Environment

Sukaviriya and Foley [1993] created an overlay of the UIDE knowledge model (described in Section 4.2.1) that serves as a user model which records information about the user's history of interaction. This overlay model can maintain statistical history of interaction, chronological history of interactions, and history of help requests.

AIDA

The user model used in Cote-Muñoz's [1993] AIDA (described in Section 4.2.1) considered three types of user knowledge in ordered to determine the class of user:

- the user's task domain knowledge;
- the user's general computer knowledge – this covers the general computer concepts that are needed to work with computers;
- the user's system-specific knowledge – this covers practice and experience specific to the software system that is rarely transferable to other systems.

Cote-Muñoz gives the example that if a user knows a lot about the task of engineering design but knows little about computers, then he needs completely different assistance than an expert in computer systems who knows nothing about engineering design.

GUMS – A General User Modeling Shell

Kass and Finin [1991] designed GUMS, a user modelling shell system, to serve a whole set of application programs. For each application GUMS kept a knowledge base of user models relevant to that application. Applications were responsible for acquiring information about the user and supplying it to GUMS to update the user model. In turn, the application queried GUMS to obtain information about the user.

4.4 Task Modelling and Plan Recognition

Task modelling attempts to represent the tasks that users will perform with the system. The various models can be differentiated into four categories [Wilson et al., 1988 in Dieterich et al., 1993]: models which analyze the knowledge content of real world tasks; models that predict difficulties from interface specifications; models which analyze the users' conceptual structures; and models which analyze cognitive activities. Dieterich et al. [1993] discuss task modelling. They indicate that despite the numerous formalisms for task modelling, these have been difficult to apply to intelligent systems because they do not handle modern interactive interfaces. For example, many are unable to describe parallelism and interrupts. In addition, the majority of these models are static and in order for them to be operational, they have to be transformed into an executable form.

Plan recognition attempts to recognize users' plans or parts of plans in order to obtain further information for adaptation or to infer new tasks [Dieterich et al., 1993]. There are a number of advantages to inferring users' plans through monitoring: context-dependent assistance and feedback can be provided; possible completions of sub-plans can be presented; default parameters can be supplied; and a more reasonable undo facility can be provided; global errors can be detected and possibly corrected. As one would imagine, recognizing plans when the task space is constrained and structured is significantly easier than in complex domains where a user's plans might consist of a hierarchy of subplans and subgoals needed to accomplish the overall goal [Carberry, 1989].

Encarnação [1997] differentiates three different categories of approaches to plan recognition:

1. *Intended plan recognition* occurs when the user is aware and actively cooperating with the recognition process.
2. *Keyhole plan recognition* occurs when the user is unaware or indifferent to the recognition process. This form of recognition requires less complex recognition mechanisms and provides less sophisticated interpretations of the user's actions.
3. *Obstructed plan recognition* assumes that users are aware of the recognition process and are actively trying to obstruct it.

4.4.1 Examples of Task Modelling and Plan Recognition:

CHORIS - The Computer-Human Object-oriented Reasoning Interface System

Tyler et al. [1991] describe an emergency crisis management system that is based on CHORIS, which is a generic architecture for intelligent interfaces. The architecture and hence the crisis management system contains a plan manager that incorporates a task model. The plan manager consists of three critical elements: a declarative representation of plans, routines for utilizing such representations to assist users in their interactions, and the reasoning ability to determine what particular goals users are trying to achieve. For example, when a user logs into the system and indicates that an earthquake has occurred, then the substep hierarchy for the *manage earthquake* task will appear. The interface assists the user in that the system can compare the user's commands and arguments to commands with the constraints on the task substep's parameter values and thereby detect global errors. In a similar fashion the interface can make reasonable guesses about appropriate default parameters for the current task step and provide these automatically for the user. At the time of writing, the third component which was to recognize the users intention from low level interactions had not been fully realized. No user testing was documented.

SAUCI – A Self-Adaptive User-Computer Interface

Tyler and Treu [1989] document a system called SAUCI which was designed to support learnability and usability. One of the ways the system accomplishes this goal is by providing task-specific guidance to the user. The system is essentially a graphical interface to the UNIX operating system. The user selects the desired high-level task from a menu and the system then provides a break-down of all the substeps necessary to complete the task and provides visual feedback as to which substep the user is currently engaged in. User testing showed that this system performed favourably for novice UNIX users.

4.5 Multimodal Communication/Natural-Language Dialog

Multimodal communication combines multiple forms of interaction which can include natural-language (spoken or input as text), gestures, and the more traditional interaction techniques: command line entry and direct manipulation. The goal of multimodal communication is to make interacting with technology more intuitive and natural. It can allow users to operate hands-free or eyes-free and can thus provide greater flexibility to the user. Despite the apparent advantages of multimodal communication, both determining appropriate situations for its application and the complexities of processing natural language remain challenging areas of research [Encarnação, 1997].

4.5.1 Examples of Multimodal Communication/Natural-Language Dialog:

CHORIS – The Computer-Human Object-oriented Reasoning Interface System

The emergency crisis management system [Tyler et al., 1991] described in Section 4.4.1 allowed the use of natural language queries and gestural input in the middle of such queries. For example, the user could type the query "What is the population of these schools?" and subsequently gesture through cursor pointing to the representations of the desired schools.

The Adaptable User Interface

Kantorowitz and Sudarsky [1989] developed a User Interface Management System that supports the creation of adaptable user interfaces which allow the user to switch between dialog modes (not strictly limited to direct manipulation and command language) at the token level of granularity.

4.6 Intelligent Help

Providing help for the user has proven to be a more difficult problem than most designers would have expected. Research has shown that online help or manuals are not users' preferred method of knowledge acquisition [Rieman, 1996]. The problem is that users have difficulty finding the help that they need: help provided is not related to the user's context, more information is given than necessary, or the information given does not match the user's expertise level. Research in intelligent help has attempted to address these problems.

4.6.1 Examples of Intelligent Help:

SAUCI – A Self-Adaptive User-Computer Interface

SAUCI [Tyler and Treu, 1989], described in Section 4.4.1, attempts to support learnability by providing intelligent advising. Part of the intelligent advising capability is giving user-tailored advice on the UNIX commands. When the user requests general help on a command, a help sheet pops up providing the purpose, arguments, and operation of the command. Each of these components of the help sheet is tailored to the user level. The user can also request specific advice by providing a command and the arguments that they would like to try. Rather than the system executing the command, the system provides feedback on any potential problems that would arise from the command execution. Tyler and Treu [1989] note that this supports exploratory learning.

UIDE – The User Interface Development Environment
Applications that are built using UIDE (described in Sections 4.2.1 and 4.3.1) are able to automatically generate and provide context-sensitive, animated, and multimedia help [de Graff et al., 1993; Sukaviriya et al., 1990; Sukaviriya et al., 1993c; all three references cited in Encarnação, 1997].

4.7 Intelligent Tutoring Systems

Intelligent tutoring (teaching) systems are concerned with the acquisition of domain knowledge rather than operational knowledge. They attempt to follow the teaching model of a good teacher who is faced with a diverse group of students. The model adapts the teaching content and style to the domain being taught as well as to the individual student's needs and abilities [Kay, 1993].

Cardiac Tutor
The Cardiac Tutor is a knowledge-based simulation for teaching about cardiac resuscitation [Woolf, 1996]. The Tutor includes both a real-time simulation and a graphical view of an emergency room patient. The students are given the goal to save the patient by selecting the proper advanced cardiac life support procedures. Clues are provided through spoken advice, emergency room sounds, and graphical representations of ECG traces, blood gases, and vital signs. Automated tutorial help is offered with cutomized problems to suit the student's level of achievement. The Tutor provides positive feedback for both good and improved performance and incorrect behaviour is categorized and commented upon. Preliminary user feedback was very positive.

4.8 Dynamic Presentation

Research in dynamic presentation seeks to provide views of data that are dependent on the characteristics of the data set and tailored to the individual user's experience, capabilities, and preferences. The goal is to make the data as comprehensible as possible.

4.8.1 Examples of Dynamic Presentation:

GAMES – Guided Adaptive Multimedia Editing System

Gutkauf [1997] introduces a chart editing system which generates critiques by user request. These critiques are based on a user model, on expert knowledge in chart editing and on the chart currently being edited. The goal is for the system to help the author to avoid commonly made mistakes and to empower recipients of charts to adjust certain parameters (e.g., colors) to their individual abilities and needs. A chart will only change its appearance or behaviour when the user requests a critique. The user can also request an explanation from the critique.

AGA – Adaptive Graphics Analyser

Holynski [1988] describes AGA which generates images that are classified based on 8 variables: balance, grid size, busyness, complexity, regularity, colour variety, shape variety and symmetry. He had 200 subjects evaluate the images and standard regression analysis was used to discover which variables were appropriate predictors for user preference. In order to determine more detailed information the average attractiveness rating for each image along with the value of the 8 variables were input into a rule acquisition program. The program produced clear presentation rules for a given set of viewers such as attractiveness is low if complexity and busyness are both low. AGA modifies rules obtained from the rule acquisition program by accepting specific knowledge from a particular user about that individual's perceptual judgment.

4.9 Software Architectures

Early research in intelligent interfaces was understandably more focused on trying out ideas rather than being concerned about the extensibility, understandability, reusability, and maintainability of the implementation that instantiated the ideas. It soon became clear that the aforementioned concerns needed to be addressed and that an understanding of the software architecture for the software components that comprised the intelligent aspect of systems was needed. This of course parallels a larger trend in computer science over the last two decades towards extensible, understandable, reusable, and maintainable systems.

The *Seeheim Model* [Pfaff et al. 1985, in Encarnação, 1997] represents one of the first and best known models for general user interface architecture. There are three main components to this model. The *presentation* describes the visual interaction objects on a lexical level. The *dialog* describes the structural elements of the dialog and the behaviour of the interaction objects on a syntactic level. The *application interface* describes the purpose of the dialog in the proper application context on a semantic level.

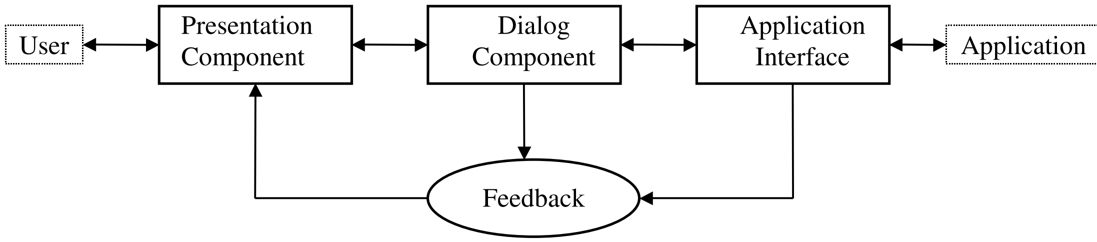

Figure 5: Seeheim model

Other well known models include the Seattle model, the Lisbon model, and the PAC model [Bass and Coutaz, 1991]. Despite the variety, there has been a lack of consensus on which is the best. This has prompted some of the leading researchers in this area to conclude that a single prescriptive model to fit all types of interactive systems is very difficult, if not impossible, to define [UIMS tool Developers Workshop, 1992].

To address this deficiency, these researchers proposed an approach that examined the nature of the data that passes between the user interface and the non-user-interface portions of an interactive system. The approach led to the definition of the *Arch Model* which models the runtime architecture of an interactive system.

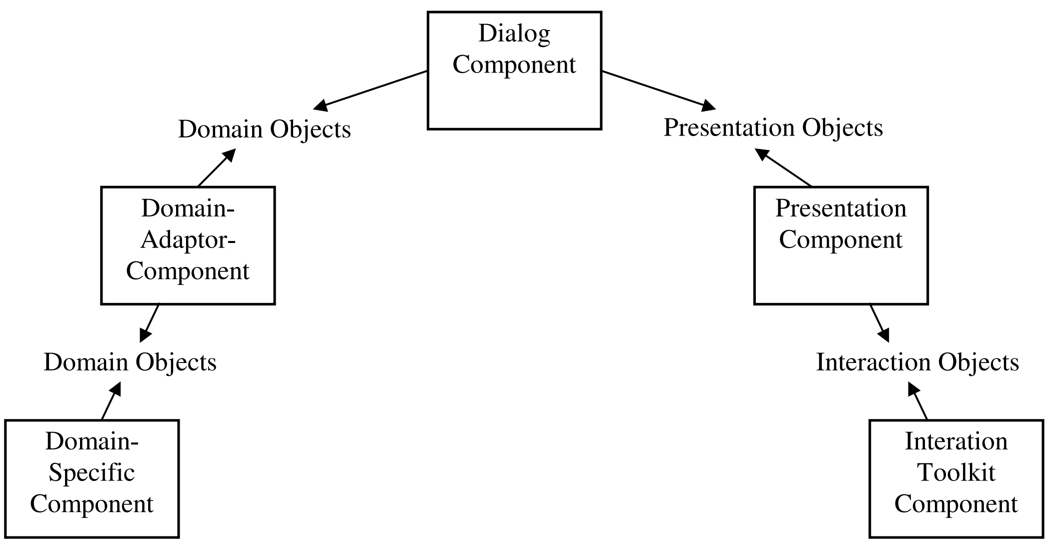

Figure 6: Arch Model [UIMS Tool Developer's Workshop, 1992].

Components of the Arch Model as described from the UIMS Tool Developer's Workshop [1992]:

- *Domain-Specific Component*: controls, manipulates and retrieves domain data and performs other domain-related functions.

- *Interaction Toolkit Component*: implements the physical interaction with the end-user (via hardware and software).
- *Dialog Component*: has responsibility for task-level sequencing, both for the user and for the portion of the application domain sequencing that depends upon the user; for providing multiple view consistency; and for mapping back and forth between domain-specific formalisms and user-interface-specific formalisms.
- *Presentation Component*: a mediation, or buffer component between the Dialog and the Interaction Toolkit Components that provides a set of toolkit-independent objects for use by the Dialog Component.
- *Domain-Adaptor Component*: a mediation component between the Dialog and the Domain-Specific Components. Domain-related tasks required for human operation of the system, but not available in the Domain-Specific Component, are implemented here. The Domain-Adaptor Component triggers domain-initiated dialog tasks, reorganizes domain data, and detects and reports semantic errors.

The objects depicted in the figure represent information that is transmitted between the components. The term "object" does not refer to formal objects as they exist in object-oriented programming. Rather, objects are an abstraction for describing a communication mechanism.

- *Domain Objects*: when used in the Domain-Specific Component, Domain Objects employ domain data and operations to provide functionality not associated directly with the user interface. In the Domain-Adaptor Component, domain data and operations are used to implement operations on domain data that are associated with the user interface.
- *Presentation Objects*: are virtual interaction objects that control user interactions. Presentation Objects include data to be presented to the user and events to be generated by the user. The medium used in the presentation or event generation is not defined.
- *Interaction Objects*: are specially designed instances of media-specific methods for interacting with the user. Interaction Objects are supplied by the Interaction Toolkit software and may be primitive or complex.

The Arch model can be generalized to the *Slinky Metamodel* [UIMS tool Developers Workshop, 1992] which essentially provides a set of Arch models by shifting functionality among components. Such a metamodel is desirable because it enables the selection of the Arch model to be dependent on the goals of the developers, their weighting of development criteria, and the type of system to be implemented.

Although the architecture models mentioned thus far have been instrumental for research in user interface software architectures, none of them are sufficient for the realization of intelligent user interfaces. To address this Hefley and Murray [1993] integrated the Triple Agent Model by Card [1989; Card 1984 in Encarnação, 1997] shown in Figure 7 with the Arch Model to form the *Arch Model for an Adaptive Intelligent Human-Machine Interface*.

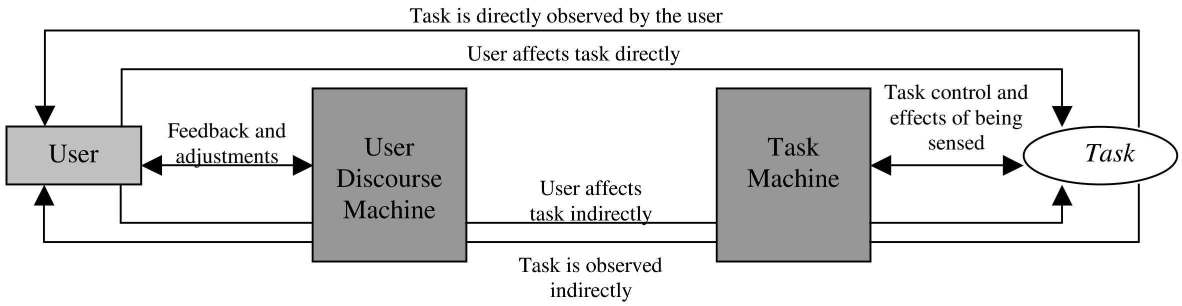

Figure 7: Card's Triple Agent Model of human-computer interaction (diagram adapted from Encarnação [1997]).

The Triple Agent Model comprises four parts: the user, a User Discourse Machine which interacts with the user, a Task Machine interacting with the task, and the task itself. Card [1989] notes that the User Discourse Machine and the Task Machine may in fact be independent computers or they may just be separate modules within the same system. There are three agents in this model: two computational agents (the Task Machine and the User Discourse Machine) and the user. Each of these agents can potentially have models of the other agents and possible even of the other agents' models.

The *Arch Model for an Adaptive Intelligent Human-Machine Interface* extends the Domain-Adaptor Component of the Arch Model to include the following three components: *Domain Adaptation*, *Discourse Management,* and an *Intelligent Decision Support System* (IDSS). The integration described by Hefley and Murray [1993] is not entirely clear but the main points are paraphrased below:

- Intelligent interfaces are really instances of domain adaptors.
- The domain specific component could house the user model data and both the domain-specific and dynamic knowledge-bases.
- Layers close to the user interface itself can perform recognition, presentation, explanation, user tailoring and so on.
- A robust intelligent interface will have several embedded layers within it, in addition to the provision of adaptivity and intelligent presentation. These include specific levels for:
    - Domain adaptation: goal management, higher level (i.e., cognitive) user modelling.
    - Discourse management: management of the interaction, action/presentation-level user modelling.

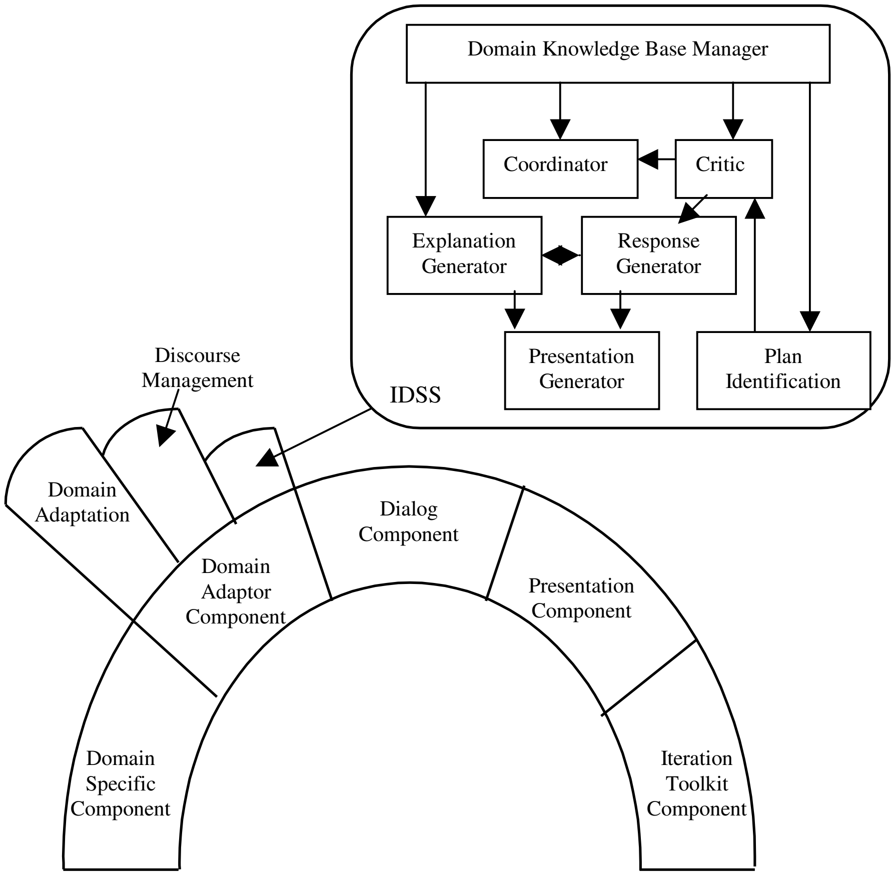

Figure 8: Architectural model for an adaptive human-machine interface [Hefley and Murray 1993].

4.10 Evaluation

The evaluation of adaptive systems has been identified as a weak area within this field of research [Dieterich et al., 1993]. Höök [1997] identifies important issues relevant to evaluating adaptive systems. The main points include:

- The importance of being able to distinguish the adaptive features of the system from the general usability of the system. Höök [1997] notes that most studies of adaptive systems are comparisons of the system with and without adaptivity and the problem can be that the non-adaptive system may not have been designed optimally for the task.
- The issue of what to measure. Often the main evaluation criteria is task completion time. Although this may be important it may not be the best or only measure. Höök describes a hypermedia system for which the quality of the search and the result is more important than the overall search time.

- Users' own evaluations of the system are important. How do they feel the adaptive and non-adaptive systems compare? Do they feel in control of the adaptive parts of the system?
- Duration of study is relevant. The advantage of most adaptive systems is that they adapt to the users' changing needs and goals. Thus short term studies that do not allow these needs to change in a natural way do not provide an accurate assessment of adaptive systems.

5 Summary and Discussion

Over the last two decades there has been a noticeable increase in the number of features that are included with application software. As previously mentioned, this trend is not isolated to software but can be seen also when considering household appliances such as food processors and VCRs. "The more the better" seems to be the prevailing philosophy. There has been some research on how people are coping with the added complexity but not sufficient research to clarify the issue. A brief summary of the research is as follows:

- Users who are new to an application are able to complete tasks faster and with fewer errors when functionality that is not needed for the tasks is blocked off.
- If a task can be completed on two different versions of a software package, a user who is new to the package will be able to complete the task on the earlier version faster.
- The number of distracting interface objects affects the ability of reasonably experienced users who are new to an application to find the appropriate object.
- Users generally use only a subset of an application's functionality and they don't often master even this subset.
- Users are generally more comfortable with mastering a subset of functionality rather than being a novice with respect to the total functionality.

The latter two points suggest a trend about usage: only a subset of an application's functionality is used by the majority of users. It is not exactly clear why this is the case. One possibility is that the full functionality is not needed by most users. Another possibility is that users don't know how to use the functionality or don't even know that it exists and so although they could use it, they get by without it. The reality is most certainly a combination of these two scenarios, but the extent to which one scenario is more prevalent than the other is unknown.

The first three points above strongly suggest that for users who are new to a system, it is significantly easier to accomplish a task if the features that are not required for that task are either removed from the interface or are blocked off. This, in and of itself, should not seem entirely surprising. One only need look for examples in the everyday world of how users learn to use and continue to use complex tools or featured *systems*. Two such examples come to mind.

One example is learning to use a calculator. Speaking from personal experience, I was given my first calculator when I was in grade school. It supported little more than addition, subtraction, multiplication, and division. When I entered high school, this calculator no longer met my needs and so I acquired a basic scientific calculator. And in second year of my undergraduate degree I transitioned yet again, but this time to a programmable calculator.

Putting the cost factor aside, no one would think of giving a grade schooler a programmable calculator because it is obviously far beyond the needs of a child. It would only confuse the child and the likelihood of the child pressing the wrong button by mistake would be high. Not only would such a calculator not fit the task of a child but it would likely hamper the child's learning.

Another example is learning to drive a car. This example isn't quite as clean as that of the calculator but is worthwhile nonetheless. To make the example work one needs to include the physical world as part of the *system* in question. Most city people when they learn to drive start out in an easy condition. They may start out in a large, empty parking lot and when they feel comfortable controlling the car they will probably try driving in a residential area where a few basic traffic signs need to be observed and other cars need to be negotiated, all at a reasonably low speed. Next, a new driver would probably feel most comfortable trying to drive on major city streets where more traffic signs are found and more complex maneuvers, such as changing lanes, are required. The new driver would most likely leave highway driving until last. Despite the minimal traffic signs on highways, the high speed usually deters new drivers from highway driving until they are comfortable in low-speed conditions.

What this example shows is that people, when given the opportunity, chose to reduce the complexity when learning a new system. It is worth pointing out that by the time a person is learning how to drive, he/she often has a very good operational model of an automobile. This is what enables new drivers to set the level of difficulty with which they are comfortable. By comparison, novice users have a poor operational model of a computer.

Returning again to the discussion of the literature, there are a number of issues and questions that remain unanswered:

- All of the literature that attempts to assess reduced complexity only looks at the first few hours a user spends with the system. How do users cope with a full-featured system beyond these initial hours? If users were to continue using the reduced system, at what point would they find it to be insufficient? Is there something in between a reduced system and a full system that would be appropriate for some users?
- Is learning on a simplified system equally beneficial for all users? One might presume that it is more beneficial for a novice user than an expert user. Is this true? To what extent? Is it dependent on whether the user is novice/expert in the computer domain or the task domain?

We do know how users approach learning a system. They learn through exploration, by applying previous knowledge, by taking formal training, by reading the user manual, by taking advantage of online help and online tutorials and demonstrations, and by asking for assistance from friends and colleagues. Although all of these learning avenues are used, research clearly shows that users generally learn through exploration and they do so in the context of real tasks. This learning behaviour is sometimes referred to as trial and error, learning by doing, or simply active learning. There is evidence that exploration is occasionally done in conjunction with reading the manual or using online help. Just because exploration is the most common approach, however, it is not to say that this is the most effective way to learn. Empirical evidence seems to suggest that this method is more effective than some passive methods, such as reading the manual or using online help and tutorials, however, there is no evidence that compares exploratory learning to formal training.

There is empirical evidence that suggests ways to make exploratory learning more effective. In general, if users are left to explore in an unrestricted fashion, they do not behave adaptively, they interact too much, and think too little. This maladaptive behaviour is in fact encouraged by the interactivity of systems. Exploratory learning can be made more effective when it is rooted in relevant tasks, and when users are encouraged to reflect on their interactions. Reflection can be enforced by a number of means, namely, by limiting the number of keystrokes users have during exploration, by making the interface more difficult to use, and by forcing users to explore one subset of the functionality at a time. Limiting keystrokes is an interesting empirical result, however, there is no obvious way to use this technique in practice. Similarly, making the interface more difficult doesn't seem like a practical long term solution especially because users clearly prefer easier interfaces (such as direct manipulation) even though they may not equally promote good problem-solving behaviours.

Forcing the exploration of one subset of functionality at a time is an extrapolation of the functionality blocking that was seen in the Training Wheels Interface [Carroll and Carrithers, 1984a, 1984b] in which there was only a single functionality subset. What hasn't been suggested by the literature is a means of implementing functionality blocking with more than one functionality subset. The Trudel and Payne [1995] study reported the benefits of limiting exploration. Their experimental design had the experimenters advance the user to a new functionality subset after a given amount of time. This, of course, is not a practical solution for commercially available software. This study as well as that of the Training Wheels Interface present some ground work on functionality blocking, but many questions remain before this strategy can be considered for commercial software:

- How can appropriate subsets of functionality be determined? To what extent is functionality layered or clustered?
- How should the user transition from one subset to another?

These are fundamental questions that need to be addressed if functionality blocking is going to be a viable method to support learning by the user. It is essential that there be some way to determine what functionality needs to be grouped, what functionality is needed all the time, what functionality is only needed occasionally, what functionality is only needed by the expert users, etc. Buxton's [1998] model seems to suggest a point of departure not so much in terms of the expert/novice dimension but in terms of functionality grouping. He advocates parsing functionality based on task although this in and of itself remains a research question.

It is worth considering a slight extension of Buxton's model such that specific tools appear within a more general tool. The representation of this can be seen in Figure 9. Although at first glance the cognitive load (i.e., the area under the curve) is higher than what is shown in Figure 3(a) it is important to remember that users use only a small subset of the total functionality. The user would only need to use those tools within the toolset that are required. The image in the figure should be understood to be significantly different than what is common in today's user interfaces. Take Microsoft Word for example. Much of the functionality is always available from the first few levels of the menus and the default toolbars. Additional toolbars can be added to both increase the accessibility of some functionality already found in the menus and also to make accessible functionality not available in the menus.

Continuing with the word processing example, the model in Figure 9 advocates that the functionality be grouped into tools that address given tasks. For example, when the user wants to draw in a document, the user is placed in the drawing tool. All functionality required for drawing operations is accessible from that tool and nothing more. This would most likely mean that there would be some overlap in functionality between tools. There would be some "net benefit" because of consistency of style and terminology among tools and also because the tools could be aware of the context in which they are operating.

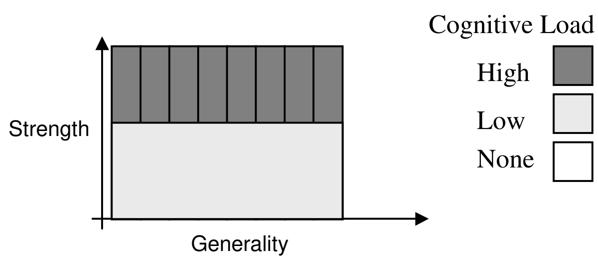

Figure 9: Extension of Buxton's model to include strength and specificity within a general tool.

Assuming that functionality subsets can be determined, then the next hurdle is figuring out how to design a system that supports multiple functionality subsets such that the user can transition easily between the subsets.

Although the ability to dynamically adjust the available functionality seems like a tall order, one needs to remember that software is fundamentally more malleable than most technology. It is precisely this malleability that is the subject of research in adaptive and intelligent user interfaces. Adaptive interfaces seek to adjust the interface such that the functionality accessible matches the user's needs, preferences and skills. Initial research in this area sought to have the system self-adapt, in other words, the system inferred what the user needed and adjusted its interface accordingly. There were two main problems with this approach. The first was that the AI techniques used to determine the user's needs and wants were not as robust as initially claimed and so system inferences were subject to considerable error. The second problem was that users need to feel as though they are in control of the system; when a system changes without a visible cause it leads the user to feel a loss of control.

The research in adaptive user interfaces is motivated by the desire to improve user performance in systems that are reasonably complex. This motivation can be identified by looking at the prototypes that attempt to recognize frequent user key sequences and then provide the user with a single command or macro that accomplishes the same result, that reorder menus or dialog boxes based on the user's usage, that infer the user's task and supply appropriate default parameters, and that provide a prompter that lists the most likely actions based on the current context.

There are a couple obvious limitations of the research in adaptive user interfaces. The first is that, with the exception of the Flexcel prototype based on Microsoft Excel [Thomas and Krogsœter, 1993], all the research is based on prototype systems. One must question the validity of research that attempts to improve performance in complex systems and uses limited prototypes to do so. Another significant limitation is the lack of reported user testing. This is disappointing given that the intelligent user interface research community initially set out to be a mix of the HCI and AI research communities. User evaluation is a fundamental principle in HCI, yet it receives minimal attention in the intelligent interface research.

In general, the reduction of complexity has not been adequately addressed in intelligent user interface research. Take the Adaptive Action Prompter [Malinowski et al., 1993] described in Section 4.2 as an example. It advocates adding an additional selection box that is always visible with the most likely options given the user's current context. The net effect of this is that more interaction objects are added to the interface in order to address complexity. This certainly seems backward. Instead of advocating better design it suggests adding another widget to navigate. The anthropomorphized paper clip in the Microsoft Office suit represents the same problem. It suggests that features can be pumped into the interface so long as there is a little agent in the corner to whom the user can ask natural language questions.

Recent literature on intelligent and adaptive user interfaces suggests a number of requirements if this research is going to impact commercial applications. These include: the need to make user modelling light-weight, and the need to put the user in control of adaptation. Research issues arise directly from these:

- A light-weight user model cannot account for all individual differences. What are the dimensions of a light-weight user model? How light-weight can a user model be before being rendered useless?

- How can the user effectively control the adaptation process?

5.1 Putting it all Together - Two Possible Scenarios

From the research gathered for this literature review it is possible to imagine at least two distinct interface scenarios that provide at least partial solutions to the functionality explosion that has occurred. Each of these is based on metaphors that are described below.

5.1.1 Gradual tool selection and de-selection - *toolbox/workbench metaphor*

A carpenter initially starts off with a clean workbench. Gradually as he/she crafts the work artifact, tools are brought from the toolbox to the workbench. Sometimes the carpenter uses a tool and then immediately returns it to the toolbox knowing that the tool will not be needed again in the near future. Other times, tools will be left on the workbench after their use in anticipation of using them again in the near future. At some points the workbench will become cluttered with tools to the point that it is difficult to continue working with the work artifact. Many tools may be returned to the toolbox in order to clear some space. The carpenter will inevitably change tasks periodically. These task changes may accompany significant changes to the selection of tools used on the workbench.

Figure 10 shows what the user might see if we take this approach. There is a single workspace and a toolbox from which the user can select the desired tool(s) and return them when no longer needed. When a tool is added to the workspace, all the functionality of the tool is made available in the workspace. When the workspace becomes too cluttered, the user puts the tool back in the toolbox.

Example: The user loads a word processing application for the first time. Only a minimal functionality subset is available, i.e., the user can create a new document, open an existing document, save, print, input text, cut, paste, and delete. The toolbox that accompanies the barebones word processor, includes tools such as *Text Formatting*, *Drawing, Charts, Tables,* etc. The user is easily able to add or remove these tools from the application's workspace.

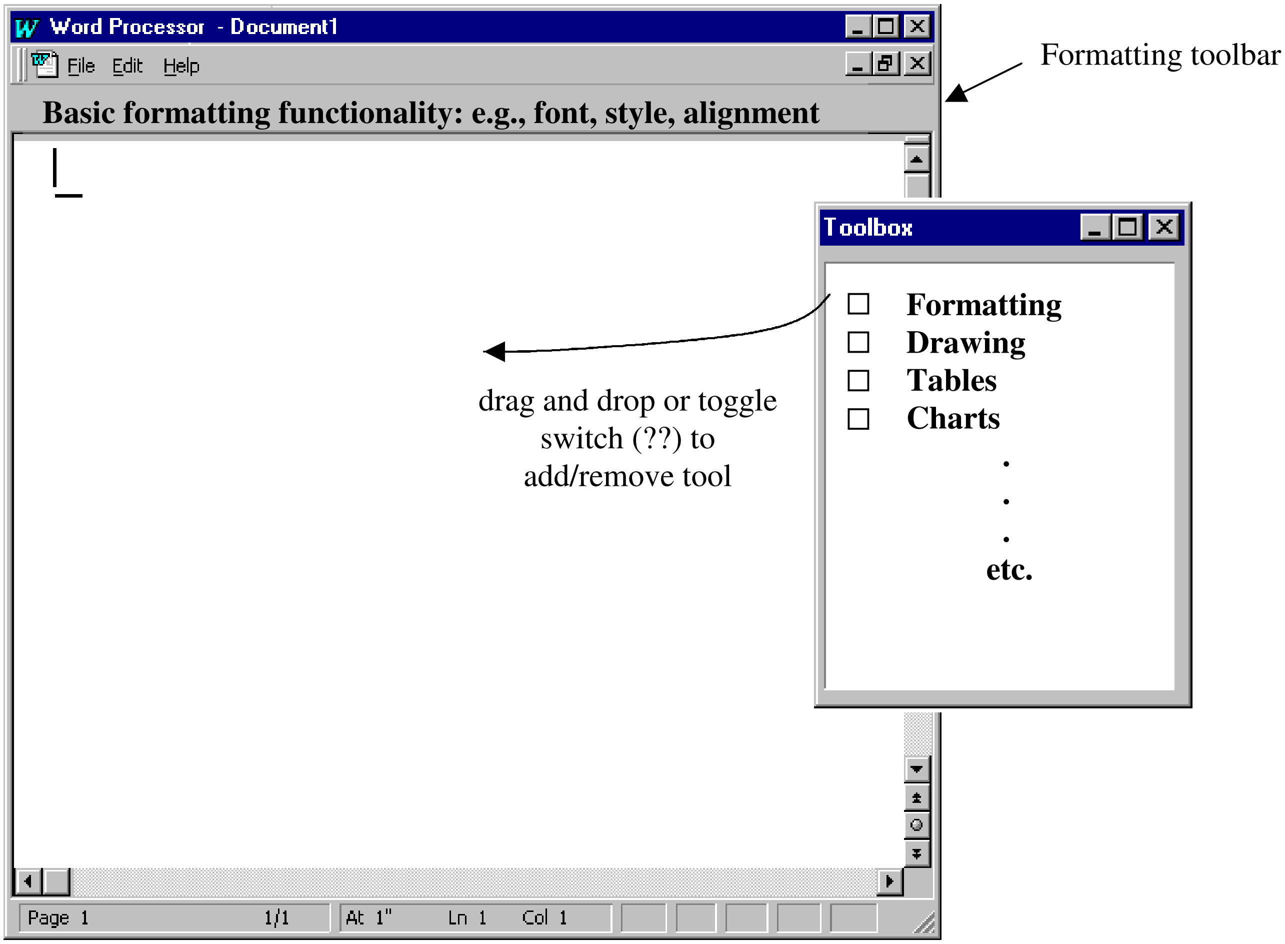

Figure 10: Workspace and toolbox described in Scenario One. The image depicts the workspace after the Formatting tool has been added.

5.1.2 Task driven tool selection - *multiple workbench metaphor*

An alternative to bringing all the required tools to a single workbench is a workshop with multiple workbenches, each providing different utility. The carpenter moves with the work artifact from one bench to another, e.g., because some tools such as a drill press or electric saw are large, certain tools require a special surface, or there are physical constraints such as proximity to an electric outlet or a light source. In this scenario it is easier to leave the tools stationary and for the carpenter to move to them.

Figure 11 shows what the user might see if we take this approach. When the user first accesses the system he/she is required to answer a few brief questions which allows the system to roughly assess the user's knowledge, i.e., light-weight user modelling categorizes users into one of three categories: novice, intermediate, or expert (or possibly a three dimensional model of expertise). The user next selects a task from the list of high-level tasks presented by the system and tools appropriate to the selected task are then automatically selected by the system. (Alternately, the user will be presented with a list of functions rather than tasks – this to be determined through user studies). The system then opens into a default tool and a gestalt click-able map representing all tools available is provided at the periphery of the workspace. In this way, this system can be adapted to both the user's current task (or function) and knowledge. For example, all of a tool's functionality will be disclosed to an expert user whereas only a subset of the functionality will be disclosed to a novice. When a novice or intermediate user is ready to have access to increased functionality within a tool they click a button that represents "More functionality please". The system would present a high level view of all the alternatives and the user would once again select a high-level task.

Example: The user loads a word processing application. Based on the user's response to a few brief questions, the system determines the user to be a novice. The user selects "create business letter" as the desired task, he/she is given the default novice tool-set which includes three tools, *Text Entry and Formatting*, *Spell Checker*, *Thesaurus.* The basic functionality of *Text Entry and Formatting* is for example, font, font size, style, and justification. To access more advanced formatting features, e.g., paragraph, page layout, etc. the user clicks the button asking for more functionality.

To select another tool, e.g., the Spell Checker, the user clicks on its representation in the map. The Spell Checker is highlighted and basic functionality available to that tool is shown in the toolbar. To return to Text Entry and Formatting, the user need only click its representation in the map.

5.2 Research Contributions

Investigating scenarios such as those mentioned above could potentially provide all or some of the following research contributions:

- a continuation of the training wheels (functionality blocking) research that was very successful and highly cited in the literature but has never been furthered;
- an understanding of the benefits of a simplified system for diverse users;
- assessment of adaptive strategies in the context of commercial, complex software rather than prototype software (e.g., MS Word, MS Excel, Lotus Notes, Alias Wavefront's Maya, MAD);
- performing user testing of an adaptive interface which would further the understanding of appropriate user testing strategies and methodologies for adaptive systems;
- an investigation of the pertinent issues of light-weight user modelling in adaptive interfaces;
- an assessment of an adaptive interface beyond the first few hours of usage;
- the development and testing of a 3D model of expertise.

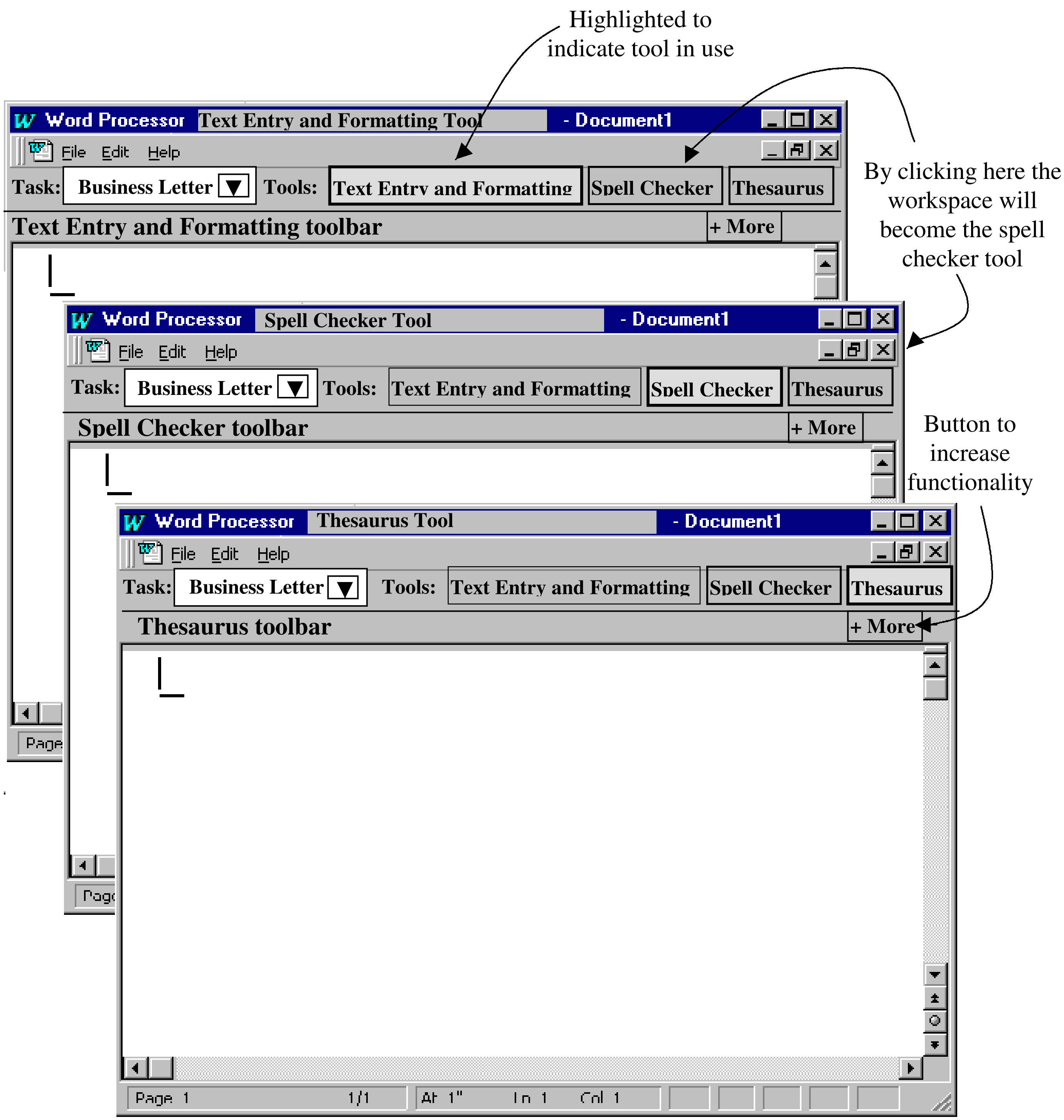

Figure 11. Workspaces as depicted in Scenario Two. At any given time only one of these screens is visible. It is possible to navigate from one screen to any other by simply clicking on the desired tool button

6 References

Baecker, R., Grudin, J., Buxton, W., and Greenberg, S. (1995). *Readings in Human-Computer Interaction: Toward the Year 2000*, San Mateo, California: Morgan Kaufmann.

Baecker R., Small, I., and Mander, R. (1991). Bringing icons to life. *Proceedings of CHI'91*, 1-6.

Baecker, R. and Buxton, W. (1987). *Readings in Human-Computer Interaction: A Multidisciplinary Approach*, San Mateo, California: Morgan Kaufmann.

Bass, L., and Coutaz, J. (1991). *Developing Software for the User Interface*, Reading, Mass: Addison-Wesley.

Boehm, B.W. (1988). A spiral model of software development and enhancement. *IEEE Computer*, 21(2), 61-72.

Bonar, J. and Liffick, B. (1991). Communicating with high-level plans. in *Intelligent User Interfaces*, J.S. Sullivan and S.W. Tyler (eds.), 129-156.

Borenstein, N. (1985). *The Design and Evaluation of On-line Help Systems*. PhD Dissertation, CMU-CS-85-151, Department of Computer Science, Carnegie-Mellon University.

Bösser, T. (1985). *Learning in Man-Computer Interaction: A Review of the Literature*, Germany: Springer-Verlag.

Buxton, W.S. (1998). Conversation with Bill Buxton, Chief Scientist Alias Wavefront and Silicon Graphics; Associate Professor, University of Toronto, Spring, 1998.

Carberry, S. (1989). Plan recognition and its use in understanding dialog. in *User Models in Dialog Systems*, A. Kobsa and W. Wahlster (eds.), Berlin, Heidelberg: Springer-Verlag, 133-162.

Card, S.K. (1989) Human factors and artificial intelligence. in *Intelligent Interfaces: Theory, Research and Design*, P.A. Hancock and M.H. Chignell (eds.), North-Holland: Elsevier Science Publishers B.V., 27-46.

Carroll, J., and Carrithers, C. (1984). Blocking learner error states in a training-wheels system. *Human Factors*, 26(4), 377-389.

Carroll, J., and Carrithers, C. (1984). Training wheels in a user interface. *Communications of the ACM*, 27(8), 800-806.

Carroll, J., and Mack, R. (1984). Learning to use a word processor: By doing, by thinking, and by knowing. In Thomas, J., and Schneider, M. (eds.), *Human Factors in Computer Systems,* Ablex, 13-51.

Carroll, J.M. (1990). *The Nurnberg Funnel*. Cambridge, MA: MIT Press.

Clement, A. (1993). Computer support for computer work: A social perspective on the empowering of end users. in *Readings in Groupware and Computer-Supported Cooperative Work: Assisting Human-Human Collaboration*, R.M. Baecker (ed.), San Francisco, California: Morgan Kaufman, 315-328.

Computer Science and Telecommunications Board, National Research Council (1997). *More than Screen Deep: Toward Every-Citizen Interfaces to the Nation's Information Infrastructure,* National Academy Press, Washington, D.C.

Constantine, L.L. (1995). *Constantine on Peopleware*. Englewood Cliffs, NJ: Prentice Hall.

Cote-Munoz, J.A. (1993). AIDA - An Adaptive System for Interactive Drafting and CAD Applications. in *Adaptive User Interfaces: Principles and Practice*, M. Schneider-Hufschmidt, T. Kuhme and U. Malinowski (eds.), Elsevier Science Publishers B.V., 225-240.

Davis, S. and Bostrom, R. (1992). An experimental investigation of the roles of the computer interface and individual characteristics in the learning of computer systems. *International Journal of Human-Computer Interaction*, 4(2), 143-172.

Dieterich, H., Malinowski, U., Kuhme, T., and Schneider-Hufschmidt, M. (1993). State of the Art in adaptive user interfaces. in *Adaptive User Interfaces: Principles and Practice*, M. Schneider-Hufschmidt, T. Kuhme and U. Malinowski (eds.), Elsevier Science Publishers B.V., 13-48.

Dryer, D.C. (1997). Wizards, guides, and beyond: Rational and empirical methods for selecting optimal intelligent user interface agents. *Proceedings of IUI'97*, 265-268.

Encarnação, L.M. (1997). *Concept and Realization of Intelligent User Support in Interactive Graphics Applications,* unpublished dissertation, der Fakultät für Informatik, der Eberhard-Karls-Universität zu Tübingen. http://www.gris.uni-tuebingen.de/gris/proj/guis/Papers/DISS/diss.html.

Fischer, G. (1993). Shared knowledge in cooperative problem-solving systems - integrating adaptive and adaptable components. in *Adaptive User Interfaces: Principles and Practice*, M. Schneider-Hufschmidt, T. Kuhme and U. Malinowski (eds.), Elsevier Science Publishers B.V., 49-68.

Franzke, M. (1995). Turning research into practice: Characteristics of display-based interaction. *Proceedings of CHI'95*, 421-428.
Franzke, M., and Rieman, J. (1993). Natural training wheels: Learning and transfer between two versions of a computer application. *Vienna Conference VCHCI'93*, 317-328.
Gong, G., and Salvendy, G. (1995). An approach to the design of a skill adaptive interface. *International Journal of Human-Computer Interaction,* 7(4), 365-383.
Goodwin, N.C. (1987). Functionality and Usability. *Communications of the ACM*, 30(3), 229-233.
Greenberg, S. and Witten, I.H. (1985). Adaptive personalized interfaces - A question of viability. *Behaviour and Information Technology*, 4(1), 31-45.
Grudin, J. (1989). The case against user interface consistency. *Communications of the ACM*, 32(10), 1164-1173.
Gutkauf, B. (1997). Accounting for individual differences through GAMES: Guided Adaptive Multimedia Editing System. *Extended Abstracts, CHI 97*, 22-27 March 1997, Atlanta, GA, 57-58.
Hefley, W.E. and Murray D. (1993). Intelligent User Interfaces. *Proceedings of the 93 International Workshop on Intelligent User Interfaces*, January 4-7, Orlando, Florida , 3-10.
Holynski, M. (1988). User-adaptive computer graphics. *International Journal of Man-Machine-Studies*, 29, 539-548.
Höök, K. (1997). Evaluating the utility and usability of an adaptive hypermedia system. *Proceedings of IUI 97*, 179-186.
Howes, A. and Payne, S.J. (1990). Supporting exploratory learning. *Human-Computer Interaction - INTERACT '90*, D. Diaper et al. (eds.) North-Holland: Elsevier Science Publishers B.V., 881-885.
Innocent, P.R. (1982). Towards self-adaptive interface systems. *International Journal of Man-Machine Studies*, 16, 287-299.
Jackson, S.L, Stratford, S.J., Krajcik, J., and Soloway, E. (1996). A learner-centered tool for students building models. *Communcations of the ACM*, 39(4), 48-49.
Jackson, S.L, Krajcik, J., and Soloway, E. (1998). The design of guided learner-adaptable scaffolding in interactive learning environments. *Proceedings of CHI 98*, 187-194**.**
Johnson, J., Roberts, T., Verplank, W., Smith, D., Irby, C., Beard, M., and Mackey, K. (1989). The Xerox Star: A retrospective. *IEEE Computer 22(9) 11-29.*
Kantorowitz, E., and Sudarsky, O. (1989). The adaptable user interface. *Communications of the ACM*, 32(11), 1352-1358.
Kass, R. and Finin, T. (1991). General user modeling: A facility to support intelligent interaction. in *Intelligent User Interfaces*, J.S. Sullivan and S.W. Tyler (eds.), 111-128.
Kaufman, L, Weed, B. (1998). User interfaces for computers - Too much of a good thing? Identifying and resolving bloat in the user interface. *Conference Summary, CHI 98*, workshop #10, 207-208.
Kay, J. (1993). Pragmatic User Modelling for Adaptive Interfaces. In *Adaptive User Interfaces: Principles and Practice*, M. Schneider-Hufschmidt, T. Kuhme, and U. Malinowski (eds.) Amsterdam, Holland: Elsevier Science Publishers, 129-147.
Keating, D. (1998). Conversation with Dan Keating, chair of Human Development and Applied Psychology at Ontario Institue for Studies in Education, University of Toronto. Spring, 1998.
Kesterton, M. (1998). Social Studies. *The Globe and Mail*, May 20th, 1998, A20.
Kerr, M.P. and Payne, S.J. (1994). Learning to use a spreadsheet by doing and by watching. *Interacting with Computers*, 6(1), 3-22.
Kozierok, R. and Maes, P. (1993). A learning interface agent for scheduling meetings. *Intelligent User Interfaces '93*, 81-88.
Kühme, T. (1993) A user-centered approach to adaptive interfaces. *Intelligent User Interfaces '93*, 243-245.
Lester, J.C., FitzGerald, P.J., and Stone, B.A. (1997). The pedagogical design studio: Exploiting artifact-based task models for constructivist learning. *Proceedings of IUI 97*, 155-162.
Lewis, C., and Norman, D. (1986). Designing for error. in D. Norman, and S. Draper (eds.), *User Centered System Design,* Lawrence Erlbaum Associates, 411-432.
Mack, R.L., Lewis, C.H., and Carroll, J.M. (1983). Learning to use word processors: Problems and prospects. *ACM Transaction on Office Information Systems*, 1(3), 254-271.
Malinowski, U., Kühme, T., Dieterich, H., Schneider-Hufschmidt, M. (1993). Computer-aided adaptation of user interfaces with menus and dialog boxes. in *Human-Computer Interaction: Software and Hardware Interfaces, Proceedings of the Fifth Conference on Human-Computer Interaction, (HCI International '93),*

Orlando, Florida, Volume 2. M.J. Smith, and G. Salvendy, (eds). Elsevier Science Publishers B.V., 122-127.

Manley, John (1998). Canada by Design Lecture Series, Knowledge Media Design Institute, University of Toronto, March 12, 1998.

Mark, W. (1991). Foreward. in *Intelligent User Interfaces*, J.S. Sullivan and S.W. Tyler (eds.), vii-viii.

McGrath, J. (1995). Methodology Matters: Doing Research in the Behavioral and Social Sciences. in *Readings in Human-Computer Interaction: Toward the Year 2000,* R. Baecker, J. Grudin, W. Buxton, and S. Greenberg, 151-169.

Meyer, T.H., and Sutherland, I.E. (1968). On the design of display processors. *Communications of the ACM,* 11(6), 410-414.

Miller, J.R., Sullivan, J.W., and Tyler, S.W. (1991). Introduction. in *Intelligent User Interfaces*, J.S. Sullivan and S.W. Tyler (eds.), 1-10.

Munk, N. (1996). Technology for technology's sake. *Forbes*, October 21, 280-288.

Myers, B.A., Potosnak, K., Wolf, R., and Graham, C. (1993). Heuristics in real user interfaces. *Proceedings of InterCHI'93*, panel, 304-307.

Nilsen, E., Jong, H., Olson, J., Biolsi, K., Reuter, H., and Mutter, S. (1993). The growth of software skill: A longitudinal look at learning & performance. *Proceedings of InterCHI'93*, 149-156.

Norman, D.A. (1986). Cognitive Engineering. in *User Centered System Design: New Perspectives on Human-Computer Interaction*, D.A. Norman and S.W. Draper (eds.), Hillsdale, N.J.: Lawrence Erlbaum, 31-61.

Norman, D.A. (1990). Human error and the design of computer systems. *Communications of the ACM*, 33(1), 24-27.

Norman and Spohrer (1996). Learner-centered education. *Communcations of the ACM*, 39(4), 24-27.

Olson, J.R., and Olson, G.M. (1990). The growth of cognitive modeling in human-computer interaction since GOMS. *Human-Computer Interaction*, 5, 221-265.

Page, S.R., Johnsgard, T.J., Albert, U., and Allen C.D. (1996). User customization of a word processor. *Proceedings of CHI 96,* April 13-18, Vancouver, Canada, 340-346.

Payne, S.J. and Howes, A. (1992). A task-action trace for exploratory learners. *Behaviour and Information Technology*, 11(2), 63-70.

Payne, S.J., Chesworth, L., and Hill, E. (1992). Animated demonstrations for exploratory learners. *Interacting With Computers*, 4(1), 3-22.

Polson, P.G., and Lewis, C.H. (1990). Theory-based design for easily learned interfaces. *Human-Computer Interaction*, 5, 191-220.

Preece, J. (1994). *Human-Computer Interaction*, Addison-Wesley.

Raskin, J. (1997). Looking for a humane interface: Will computers ever become easy to use? C*ommunications of the ACM*, 40(2), 98-101.

Resnick, L.B. (1990). *Knowing, Learning, and Instruction: Essays in Honor of Robert Glaser*, L.B. Resnick (ed.), Hillsdale, New Jersey: Lawrence Erlbaum.

Rich, C. and Sidner, C.L. (1996). Adding a collaborative agent to graphical user interfaces. *Proceedings of UIST'96*, Seattle Washington, USA, 21-30.

Rich, E. (1989). Stereotypes and user modeling. in *User Models in Dialog Systems*, A. Kobsa and W. Wahlster (eds.), Berlin, Heidelberg: Springer-Verlag, 35-51.

Rieman, J. (1996). A field study of exploratory learning strategies. *ACM Transactions on Computer-Human Interaction*, 3(3), 189-218.

Rieman, J., Young R., and Howes, A. (1996). A dual-space model of iteratively deepening exploratory learning. *International Journal of Human-Computer Studies*, 44, 743-775.

Rosson, M. and Carroll, J.M. (1996). Scaffolded examples for learning object-oriented design. *Communcations of the ACM*, 39(4), 46-47.

Sagar, I., Hof, R.D., Judge, P. (1996). The race is on to simplify: Pulling the unwired masses into the information age means gadgets must be as easy to use as the telephone. *Business Week*, June 24, 1996, 72-75.

Sellen, A., and Nicol, A. (1990). Building User-centered on-line help. in B. Laurel (ed.), *The Art of Human-Computer Interface Design*, Addison-Wesley, 143-153.

Schank and Kass (1996). A goal-based scenario for high school students. *Communcations of the ACM*, 39(4), 28-29.

Shneiderman, B. (1995). Perspectives: Looking for the bright side of user interface agents. *Interactions*, (January), 13-15.

Shneiderman, B. (1997a). *Designing the User Interface: Strategies for Effective Human-Computer Interaction.* Third Edition. Addison Wesley.

Shneiderman, B., (1997b). Direct manipulation for comprehensible, predictable and controllable user interfaces. *Proceedings of IUI '97*, 33-39.

Shneiderman, B., and Maes, P. (1997). Direct manipulation vs. interface agents: Excerpts from debates at IUI 97 and CHI 97. *Interactions*, (November/December), 42-61.

Soloway, E., Guzdial, M., and Hay, K.E. (1994). Learner-centered design the challenge for HCI in the 21st century. *Interactions*, (April), 36-48.

Soloway, E., Jackson, S.L., Klein, J., Quintana, C., Reed, J., Spitulnik, J., Stratford, S.J., Studer, S., Eng, J., and Scala, N. (1996). Learning theory in practice: Case Studies of Learner-Centered Design. *CHI 96*, 189-196.

Stephanidis, C., Karagiannidis, C., and Koumpis, A. (1997). Decision making in intelligent user interfaces. *Proceedings of IUI '97*, 195-202.

Sukaviriya, P., and Foley, J.D. (1993). Supporting adaptive interfaces in a knowledge-based user interface environment. *Intelligent User Interfaces '93*, 107-113.

Sullivan, J.S., and Tyler, S.W. (1991). *Intelligent User Interfaces*, J.S. Sullivan and S.W. Tyler (eds.), New York: ACM Press.

Svendsen, G.B. (1991). The influence of interface style on problem solving. *International Journal of Man-Machine Studies*, 35, 379-397.

Thomas, C.G. and Krogsoeter, M. (1993). An adaptive environment for the user interface of Excel. *Intelligent User Interfaces '93*, 123-130.

Trudel, C.I. and Payne, S.J. (1995). Reflection and goal management in exploratory learning. *International Journal of Human-Computer Studies*, 42, 307-339.

Tyler, S.W., and Treu, S. (1989). An interface architecture to provide adaptive task-specific context for the user. *International Journal of Man-Machine Studies*, 30, 303-327.

Tyler, S.W., Schlossberg, J.L, Gargan Jr., R.A., Cook, L.K., and Sullivan, J.W. (1991). An intelligent interface architecture for adaptive interaction. in *Intelligent User Interfaces*, J.S. Sullivan and S.W. Tyler (eds.), 85-109.

UIMS Tool Developers Workshop (1992). A metamodel for the runtime architecture of an interactive system, *SIGCHI Bulletin*, 24(1), 32-37.

van Oostendorp, H., and Walbeehm, B. (1995). Towards modelling exploratory learning in the context of direct manipulation interfaces. *Interacting with Computers*, 7(1), 3-24.

Vaubel, K.P., and Gettys, C.F. (1990). Inferring user expertise for adaptive interfaces. *Human Computer Interaction*, 5, 95-117.

Wahlster, W., and Kobsa, A. (1989). User Models in Dialog Systems. in *User Models in Dialog Systems*, A. Kobsa and W. Wahlster (eds.), Berlin, Heidelberg: Springer-Verlag, 4-34.

Wright, P. (1983). Manual dexterity: A user-oriented approach to creating computer documentation. *Proceedings of CHI '83*, 11-18.

Woolf, B.P. (1996). Intelligent multimedia tutoring systems. *Communcations of the ACM*, 39(4), 30-31.