A Survey of Automated Consulting in Interactive Programming Environments

Ursula Wolz
Department of Computer Science
Computer Science Building, Columbia University
New York, NY 10027

CUCS-262-87

Abstract

This paper surveys the Computer Science Literature on consulting in interactive programming environments. Three types of consulting behavior are described: systems that provide relief from mundane detail; systems that provide information; and systems that tutor new skills. Two themes run through research on all three types of systems. First, a distinction is drawn between an expert and a novice user. Most systems are geared for one or the other but not both. Second, current research on all three types indicates a need for taking users' goals into account. Systems must include consulting behavior that goes beyond the surface or syntactic level.

This research was supported in part by the United States Army Research Institute under contract MDA903-85-0103.

Table of Contents

List of Figures

List of Tables

1 Introduction

> consult *v.i.* to seek information or instruction from; ask the advice of; refer to;
>
> consultant, *n.* 1... 2. a person who gives professional or technical advice.
>
> Webster's New World Dictionary

Despite efforts to make human/computer interaction simple and straightforward, the task of programming is still a complex, knowledge intensive activity. Programming frequently requires the services of a consultant who provides technical advice. The consultant may suggest techniques for accomplishing tasks, introduce new information and procedures, clear up misconceptions, and even write portions of programs to relieve the programmer of unnecessary detail. An automated consultant would be a system that performs all or some of these functions.

Within Computer Science there are currently two approaches to helping programmers. *Relief from mundane detail* is often provided directly within programming environments. On-line help facilities are intended as *sources of information* that remind the programmer about how to do something. The approaches are not mutually exclusive. In fact, it is the subtle interaction of the two that makes an environment helpful. A third aspect of consulting behavior is *tutoring new skills*. Although programmers may learn about an environment while using it, research on how to incorporate explicit tutoring into a programming environment is still in very early stages.

This paper will survey the literature on systems that exhibit consulting behavior. The work to be discussed does not fall cleanly within the domain of a single sub-field of computer science. It draws from research on programming environments, on-line help systems, and intelligent tutoring systems.

A major theme that emerges from this research is the fundamental difference between helping a novice and an expert. It is often claimed that novices crave simplicity while experts will sacrifice it to achieve greater flexibility and control. Most of the systems assume an audience of either novice or expert users, but not both. This simplifies the task of providing appropriate help, but introduces a problem for users. At some point they must "graduate" from being novices and learn to deal with programming environments as experts.

Another important theme concerns the focus of the consultation. Early work from all three fields only gave programmers help on a surface level. The emphasis was on preventing programmers from violating syntactic rules, informing programmers on proper syntactic forms, and noticing inappropriate steps in simple procedures. As these systems were used, it became clear that programmers required assistance at a level that took their intentions or goals

into account. Consultation at a surface level simply informs programmers about whether their actions can be executed within the environment. It is unable to recognize whether programmers' actions will accomplish the task at hand. Surface level help therefore has the potential to misinform, since a syntactically correct action may be inappropriate to a programmer's goal. Furthermore, empirical studies [Kay & Black 85] suggest that expert programmers rely on previously successful plans for accomplishing goals, and that they adapt familiar methods to new problems. A good human consultant can interact with an expert on a goal/plan level. Current research on consulting is beginning to address how systems themselves can exhibit such behavior.

Section 2 explores the contribution of work on programming environments. The major emphasis is on relief from mundane detail involving bookkeeping, syntactic expressions and goal satisfaction. *Managerial supervision* relieves the user from bookkeeping tasks related to file-handling and system maintenance. *Non-textual interaction* is provided through *Structure Editors* and *Viewing Systems*. The former impose a structure that helps prevent superficial mistakes caused by poor typing or spelling. The latter provides extremely concrete visual representation of abstract entities. The idea of a programmer's apprentice that has knowledge of programming *cliches* is a first step toward providing relief on a deeper conceptual level.

Section 3 reviews research related to on-line help. This work attempts to give explicit information about commands and features of a system to experienced programmers. Most current help systems use simple methods such as *command languages* and *keywords* or *menus* to access canned text. Current help systems tend to be too complex for novices and too limited for experts. Experimental systems are being developed to meet the needs of different users through *multiple access methods* and *customized examples*. These still place the burden of locating the appropriate help on the user. Systems that infer the user's goal attempt to address this problem.

Section 4 discusses contributions from work on intelligent tutoring systems. Although the work does not always pertain directly to programming environments, the tutoring techniques presented may prove to be critical to an effective consulting system. In order to be truly helpful, a consultant system must be able to provide information in a context that the programmer can understand. It must know *how to diagnose user misconceptions* and offer appropriate remediation. It must also possess a tutoring strategy such as *coaching* or *guiding*. Coaching is loosely defined as opportunistic tutoring in open-ended environments. Guiding carefully structures the progress of a user in more controlled settings. Robust intelligent tutors are extremely hard to build, partially because they are unable to determine *why* users typically make mistakes; that is where their goals and plans might be faulty. Recent work has focused on diagnosing faulty plans.

Section 5 summarizes and contrasts the issues that are introduced in the other sections. Based on this

discussion, it will suggest a research agenda for consulting in programming environments.

2 Relief from Detail: The Contribution of Programming Environments

This section introduces the concepts of "experimental environment" and structured growth. It discusses how they led to systems that place the burden of accuracy on the system rather than the programmer. Three approaches are discussed:

- **Managerial Supervision** provides basic bookkeeping facilities such as file maintenance. It is best illustrated by Interlisp [Teitelman & Masinter 81] and the Gandalf project [Habermann & Notkin 86].
- **Non-textual interaction** reduces the likelihood of and allows recovery from typographical errors. DWIM and the Programmer's Assistant [Teitelman 84a] are both components of Interlisp. They introduce the notion of a "smart interface" that filters errors in the text strings typed by the programmer. Two conceptual perspectives have developed that bypass text parsing altogether. **Structure Editors** are best illustrated by The Cornell Synthesizer [Teitelbaum & Reps 81] and the ALOE editors of Gandalf [Ellison & Staudt 85]. **Viewing Systems** will be illustrated by Smalltalk [Goldberg 84] and Boxer [diSessa & Abelson 86].
- **Plan Level Assistance** lets the programmer concentrate on the problem, rather than the details of how to express it. The programmer can think about algorithms and abstract data structures, rather than expressions that implement those structures. The Programmer's Apprentice Project [Rich & Shrobe 78] introduced the notion of representing programming knowledge in terms of plans that accomplish computational goals.

2.1 The Need for "Environments"

Programming has always been an exacting discipline. Yet since programmers are human, they can be sloppy and forgetful. As the size and complexity of programming projects increased during the 1950's and 60's, it became clear that a design discipline was necessary to compensate for human frailty [Brooks 75].

The movement toward structured programming was a result of this need [Dahl *et al.* 76; Dijkstra 76]. Programmers were admonished not to dive into a programming project. They were asked to follow procedures for developing a well structured and modular specification. Programs were to be implemented in a step-wise fashion and debugged in a systematic and logical manner. Yet the clean design principles embodied in this approach often appeared too constraining and rigid. Furthermore, despite the best efforts, programmers still made costly mistakes at all phases of development. Not only were they responsible for finding solutions to new and hard problems, they were required to attend to myriad bookkeeping details. The interfaces they used had limited views of both the programs they were developing and the systems they were using for the development.

At about the same time, the Artificial Intelligence (AI) community began to develop research aids around the programming language LISP. Due to the nature of their work, they had an entirely different perspective on programming. Problems in AI initially tend to be poorly defined and solutions are hard to prove theoretically

correct. The terms "exploratory programming" [Sheil 84] and "structured growth" [Sandewall 78] were coined to describe project development that begins with only rough specifications. Exploratory implementations of a program allow the programmer to gain insight into the nature of the problem. A solution grows through a structured cycle of hypothesize, implement, and determine the successes and short comings. These steps lead to a modified theory and new implementation.

Structured growth encourages individual programming styles. It provides a large degree of flexibility, including the ability to write and debug portions of a program independently. A compile cycle optimized for efficiency of an end product severely restricts this method. Debugging also increased in importance, which led to the need for interactive facilities that aid this phase. AI often takes this perspective for granted. Sheil argues that exploratory programming as an initial step could lessen the rigid nature of structured programming. Although the end product should still be the result of a deliberately constructed design, the design itself could benefit from a few iterations of structured growth.

The focus of exploratory programming is on the nature of the problem, not on the implementation details of the solution. The language used is important, but not as critical as the *environment* in which the programmer must work. Any collection of tools that allows a programmer to create, edit, compile, execute and store programs can be called a programming environment. This misses the point though. The environment should be conducive to iterative phases of experimentation. It should relieve the programmer of details of system management through an interface that is informative and easy to use. The programmer should be free to concentrate on the problem, not on how to solve it using specific tools.

2.2 Managerial Assistance

Managerial Assistance is loosely defined as the ability to relieve the programmer from the details of program organization and maintenance. Such facilities are especially important when teams of programmers build large complex systems. The environment rather than the programmer is responsible for mundane bookkeeping chores. It might keep track of where particular functions and data objects are referenced and defined. It might also maintain information on recent revisions to portions of the program. Subsystems of the Interlisp and Gandalf projects illustrate these types of facilities.

2.2.1 Interlisp's Bookkeeping Facilities

Interlisp is a large LISP-based integrated system. It was developed to explore programming environment requirements for huge, organic[1] AI programs. The designers assumed that the users of Interlisp would be experts who would prefer sacrificing simplicity for sophisticated tools. Their goal was to accommodate a variety of programming styles and let the system do as much of the work as possible.

The development of the system itself is an example of structured growth. The concept of programming environment was not clearly defined when the project began. Consequently, Interlisp evolved from a collection of loosely integrated tools that the developers found useful. From the onset the distinction between primitives and user-defined functions was blurred. This led naturally to a pervasive philosophy of extensibility. A tool built to manipulate the primitives of the system ought to be able to manipulate user defined functions.[2] The File Package and Masterscope are tools providing managerial assistance that evolved in this manner.

The File Package is a collection of procedures and data objects for maintaining components of a developing system. The programmer is relieved of bookkeeping details including:

- Remembering the location of definitions of functions and data objects in files.
- Determining whether files require updating due to changes made during the session.
- Saving and restoring the state of the system under development between sessions, including maintaining the most recent value of data objects.

The programmer can explicitly call any of the functions that accomplish these tasks. The File Package can also operate independently of the user. For example, the command MarkAsChanged is invoked when an object is in need of updating. The command locates all files that contain the object and prepares them for updating. The programmer can invoke this command explicitly. The command is also embedded in system components that change objects, such as the editor, the DEFINE function and DWIM (which will be discussed in section 2.3.1). Therefore, by modifying an object in the editor, the appropriate files are automatically marked as changed.

Another form of relief from detail provided by Interlisp is embodied in Masterscope. This is an interactive program for analyzing and cross-referencing user programs. It addresses the problem of predicting the effects of proposed changes to large systems. Masterscope is used to analyze the relationship between objects. For example a change to a low level function may significantly affect a set of higher level functions that call it. Masterscope

---

[1]The term "organic" has been used to describe systems that develop through an evolutionary rather than carefully designed process.

[2]UNIX [Dolotta *et al.* 84] is another environment with a similar history and resulting philosophy. It has an enthusiastic following due to its flexibility and extensibility.

enables the programmer to locate, analyze and manipulate the calling functions in order to determine the affect of the change.

The primitive operations of the File Package and Masterscope were made accessible so that programmers could modify and extend them. Most users may never find a need for this. Occasionally though, the standard implementation will constrain some programmers. Interlisp's extensibility does not require them to abandon powerful support tools.

It is unlikely that programmers could outgrow Interlisp's sophisticated features. On the other hand, [Teitelman & Masinter 81] admit that it is not easy to learn to use its facilities, and that mastery is difficult. They argue that this is the price paid for power and productivity.

Part of the problem may lie in the nature of its development. Its organic growth was necessary since no one knew at the onset what programmers might need. The nature of its development may have contributed to a level of complexity that is not easily transmitted to new users. Early users may have developed their own conceptual models for how the pieces of the system fit together. A consistent model may not be conveyed to new users. Instead, they may be exposed to a "bag of tricks." The need for a cohesive abstraction that embodies all aspects of the system will be discussed again in later sections.

2.2.2 The Gandalf Project Relies on a Consistent Conceptual Model

Gandalf [Habermann & Notkin 86] is another project that studied the development of programming environments. It emphasized the *automatic* generation of such systems, and therefore made a strong attempt to formalize the necessary mechanisms. A distinction is made between the Gandalf development system and the environments that were generated by it. The development system is used to design and implement special purpose programming environments, including some of the subsystems of the development system itself. Gandalf developed through a process of principled design and structured growth. As such, it illustrates the design and implementation cycle that all Gandalf environments attempt to support.

The fundamental assumption of the project was that a developing system should be viewed as a collection of clearly defined data types and associated operators. By imposing this perspective at all stages of development, the environment both constrains and supports the programmers. Adhering to the design principle becomes a part of the act of programming, not merely something to which casual lip service is paid. Explicit bookkeeping notations are made along with code development and modification. Once the environment has some information about an object and its operators, it can update and modify that information automatically.

The SDC (System Development Control) is a managerial component of Gandalf that illustrates these ideas. It relies on four basic object types for maintaining information about a developing system. The objects types are project, source file, log file and access control list. A project is a collection of related source files. A source file is a small, manageable set of related data or code. A log file and access control list is associated with each project. The log file contains information on changes made to the project's source files. The access control list contains information on who may access particular source files during the development of the project.

SDC commands that manipulate these objects can be used directly by the programmer, or may be automatically invoked by the system. For example, to modify a source file for a project, a programmer uses the *reserve* command. The command automatically checks the access control list to see if the programmer has modification privileges for that file. If so, it prohibits any other team members from modifying that file at the same time. After making changes the programmer must decide whether to use either the *deposit* or *release* commands. If the changes are to be kept, deposit saves and backs up the file, and makes notations to the log file including who made the changes and when. It also prompts the programmer to create a message that describes the changes. The programmer uses the release command to abandon the modifications.

The Gandalf development system, like Interlisp, is for expert programmers. Users are expected to have sufficient expertise to make efficient use of the system. The flexibility and transparency of Interlisp encourages maverick behavior, provided one has developed the skills to customize the environment. Gandalf, on the other hand, imposes what may be a necessary discipline on its users. It presents a conceptual perspective that may constrain software developers, but at the same time guide them. It is also important to note that the Gandalf development system is UNIX-based. If all else fails one can "turn off" Gandalf, and continue directly in UNIX.

The other important distinction to make is in the environment itself. Interlisp is intended to be all things to all programmers. Novices may make limited use of all of its features. Experts can modify it to meet their needs. Gandalf, on the other hand, is a carefully layered system. It is presumed that novices begin in subsystems, and may or may not eventually make use of the Gandalf development system itself. This fundamental difference in approach will appear again in the discussions that follow.

2.3 Non-textual Interaction

Communicating with a computer is often separated into two distinct activities. One can execute commands interactively by typing commands at the prompt to an operating system for example. One can also organize a set of instructions into a program for repeated use. These activities seem to have different purposes. The former is transitory. One types a command and it may be accepted and executed. If it is rejected an error message may be

reported. The latter is more permanent since it is usually stored in some form. At first glance, both of these activities seem textual in nature. One types a command at the prompt, or creates a text file in a high level programming language. Both of these activities require the computer to translate a sequence of characters into an action. The fundamental difficulty with a textual form is that the programmer is responsible for remembering syntactic and typographic detail.

Methods of non-textual interaction have been developed to address this problem. An outcome has been to begin to blur the distinction between the *interactive* and *programming* modes described above. Three kinds of non-textual interaction have developed:

- **Spelling checkers and command recovery mechanisms** are text based, but allow the programmer to recover quickly.
- **Structure editors** protect the user from many syntactic errors by supervising the generation of code as it is entered.
- **Viewing systems** bypass the need for typing most commands. They also reduce the level of abstraction by making it extremely easy to examine and manipulate complex data and program structures.

2.3.1 A Spelling Checker and Simple Recovery Mechanism

DWIM (Do What I Mean) and the Programmer's Assistant [Teitelman 84a] are components of Interlisp that illustrate how an intelligent interface adjusts to syntactically incorrect commands. The intention of these systems is to create a buffer between the programmer and LISP. They parse instructions as they are entered and include mechanisms for analyzing and storing user input.

DWIM is best known as a spelling checker that tries to guess what the user had in mind. It attempts to match every word typed by the programmer with a known command or datum. It uses knowledge about common spelling mistakes such as transpositions, doubled characters and shift mistakes. Like other components of Interlisp it is extensible and is able to operate on new words defined by the user.

The Programmer's Assistant (PA) is the actual interface between the programmer and the various facilities of Interlisp. It keeps every command typed by the user on a history list, and refers unrecognized input to DWIM. When a command is recognized, the PA also places an object called an event on the history list. An event describes the actions associated with a command. The user can manipulate events on the list with commands such as USE, FIX, REDO and UNDO. The PA also includes facilities for extending the command set.

DWIM makes uses of the history list to determine the context for a command. For example, an unrecognized input could be permuted into a number of different commands. DWIM can refer to the current context to choose the most likely candidate. It can simply tell the programmer of the proper correction, or use the facilities of the PA to

FIX the command itself. The rational is that for common typing mistakes, the programmer would prefer to have complete automatic correction facilities. If this proves wrong in a particular instance, the programmer can always UNDO the process initiated by DWIM. The automatic components of both DWIM and the PA can be turned on and off by the programmer.

Although these systems provide invaluable help to the programmer, they seem geared to fixing a problem that ought not to appear in the first place. Simple typing and syntactic errors occur because the programmer's input mode is textual. If the command could be entered in some other form, the mistakes that DWIM and the PA catch might never occur.

2.3.2 Structure Editors

The goal of structure editors[3] is to relieve the programmer from remembering and typing syntactic details. The frequency of typographic errors is reduced because the programmer enters far less text. A program is entered and modified by manipulating language constructs rather than simple text strings. Movement through the code is usually based on hierarchical components of the structure, rather than character position. Most program construction becomes a matter of choosing the appropriate structure, and filling in the details specific to the particular program. Attempts to violate the syntactic rules of the language are either disallowed, or at the least, pointed out by the system.

The manner in which the programmer actually edits code depends both on the sophistication of the display technology and the perspective on programming taken by the developers. Older systems tend to rely on command languages that assume a line by line teletype interaction. Newer systems make use of screen displays where cursor movements control the editing process. Almost all of these editors also include mechanisms for inserting templates of general program constructs which can then be expanded. Table 1 compares the editing features of a number of structure editors.

Interlisp probably contains the oldest structure editor [Sandewall 78]. As such, it relies on a command language that allows explicit reference to LISP s-expressions. The commands and their parameters must by typed by the programmer, and therefore afford only minimal relief from syntactic error[4]. The Interlisp editor was designed to focus editing on the structural component itself, rather than on the line and character position at which it occurred in a text file. DWIM and the Programmer's Assistant evolved in an attempt to solve the typing problem.

---

[3] Structure editors are also called syntax-directed editors or language-oriented editors.

[4] Although both DLISP [Teitelman 84b] and DED [Barstow 84] provide display enhancements, these extensions to Interlisp still rely on command languages.

SYSTEM	*Teletype/ Screen*	*Use of Templates*	*Extensi- bility*	*Language Support*
Interlisp	Teletype	No	Highly	LISP - Could support others through macros
EMACS	Screen	Programmable	Highly	Can program to edit any language
Cornell Program Synthesizer	Screen	Yes	No	Generator for any language
MENTOR	Teletype\ Screen	No	No	Just Pascal
Gandalf ALOEs	Screen	Yes	Through calls to routines written in C	Designed explicitly for use with any language

Table 1: Summary of Structure Editors

Like other components of Interlisp, the editor is completely extensible and integrated with the rest of the system. Like those other components it is also difficult to master.

EMACS [Stallman 81] is not normally classified as a structure editor, but deserves mention here. Although EMACS is text based, it is a powerful, customizable screen editor. As such it can be modified to manipulate code from a structural rather than a textual perspective. For example, when operating in LISP mode, s-expressions are automatically formatted. EMACS also catches and reports any attempt to inappropriately open or close an expression. Sophisticated features such as template generation can be constructed for any language by writing macros. EMACS is completely programmable provided one is willing at times to delve into the underlying language such as TECO or LISP. EMACS attempts to be all things to all programmers. Like Interlisp, it is often extremely hard to master.

Both Interlisp and EMACS try to resolve the basic conflict between the abstract structure of a program and its textual representation. Interlisp has an underlying representation that is structural, and must work hard at providing textual representations of it to the user. EMACS, on the other hand, assumes a textual basis. It must therefore provide massive, and at times awkward facilities to interpret and analyze that text as structure. In both systems it is

no small task for a user to figure out how to manipulate structure. By the time one becomes proficient with either editor, one has probably developed the expertise to create the correct syntactic expression without support.

The Cornell Program Synthesizer (The Synthesizer) [Teitelbaum & Reps 81], MENTOR [Donzeau-Gouge *et al.* 84] and the Gandalf ALOE editors [Habermann & Notkin 86; Ellison & Staudt 85] attempt to focus programmers' attention on program structure rather than textual details. They have an underlying structural representation. MENTOR suffers from the same kinds of interactive problems as INTERLISP, since it was intentionally developed as a command language. The rational at the time was portability.

The Synthesizer and the Gandalf ALOE editors contain very similar structural representations and interactive facilities. They differ in the rigidity of the interface and command set. The Synthesizer was designed to encourage a structured programming style among novice programmers of PL/CS and Pascal. It has a static interface and fixed command set geared to this purpose. The Synthesizer has been expanded into a Synthesizer Generator through which editors for any language could be built. The Gandalf editors are a result of research on *creating* environments. All of the editors, including the one used to create other editors, use a tree structured representation of a program and a systematic, partially automated implementation procedure. Therefore, interfaces and extensions to the command set are fairly easy to generate, as are editors for new languages.

Both the Synthesizer and the Gandalf editors allow users to generate templates of syntactic structures on command. Templates include text that represents syntactically correct code, and placemarkers indicate subcomponents that must be expanded. For example, a while loop can be added to a portion of a Pascal program by invoking the template in Figure 1. Placemarkers appear in italics. Both systems allow cursor controlled movement only to placemarkers and the beginning of structures. In Figure 1 the cursor would move from the "W" of WHILE to the "e" of expression, or to the "s" of statement. Actions taken at any of these locations depend on its structure. For example, at the "e," the programmer must insert an expression, but any Pascal statement could be inserted at the "s."

```
WHILE expression DO
        statement;
```

Figure 1: Example of a Structure Editor Template

From a structured design standpoint, editing in this manner is ideal. The programmer is not only relieved of syntactic detail, but is guided through a process of step-wise refinement. The practical situation is not always as desirable. Simple textual reorganization of code becomes cumbersome if not impossible. At the moment of its

insertion, an expression has an associated syntactic structure. It can be moved to another location of the same structural type. However, it can not be modified or moved to a location of a different type. For example, a Pascal programmer might want to turn a complex assignment statement into a conditional expression. Textually all that might be required is changing ":=" into "=", which a rigid structure would not allow. Other representations such as lexical tokens [Kaiser & Kant 85] create compromises between text and structure, but a basic problem still remains. The programmer must keep in mind the abstract concept that is associated with the physical or textual representation embodied in the token.

2.3.3 Viewing Systems

Viewing systems focus entirely on concretizing abstractions. The two best examples are Smalltalk [Goldberg 84] and Boxer [diSessa & Abelson 86][5]. Within these systems it is impossible to separate the representation of data and processes from the objects themselves. Since they can be viewed, they can be straightforwardly manipulated. One interacts with the system by browsing through viewports using a pointing device or, where more appropriate, through a command language.

Although they have much in common with structure editors, the intent of viewing systems is more liberal. Programming is not seen as the systematic development of syntactic expressions. It is viewed as the act of directly manipulating data in whatever way one pleases. Programmers are encouraged to modify or "reconstruct" the environment to suit their own purposes. They can work "top down" by using established methods of systematic refinement. They can also work "bottom up" by noticing similarities between various objects and synthesizing and generalizing them.

Smalltalk is best known as the first modern example of an *object oriented programming language*. It reverses the perspective on data and process [Goldberg & Robson 83]. Most other programming languages define data within a process. Programming in Smalltalk begins by defining the information that belongs to a class or type of object. One then defines how that information relates to information of other classes of objects. Process definitions are completely dependent upon the class. Processes are methods for sending and receiving information between objects. In other words, they are simply ways to send *messages*. A specific data item is constructed by creating an *instance* of a class.

The information that an object possesses and the methods it uses can be modified without affecting other classes of objects. For example, all classes of objects have methods for displaying their information. Modifying a

[5]Many of the Structure Editors discussed in the previous section also include aspects of viewing system that rely heavily on concepts first developed in Smalltalk.

display message for one class does not affect either the information or the display messages associated with any other class.

The object's perspective in and of itself only begins to concretize the process of programming. It is the direct *visual* manipulation of objects through *views* that separates Smalltalk from other languages that have adopted object oriented programming. Views provide systematic mechanisms for displaying and interacting with objects. More importantly, they are an integral part of the definition of all objects.

Four types of views exist, and each is an object that can be modified:

1. A **MENU** is an external view of an object that indicates what messages the object can receive. It reminds the user of what actions can be taken on the object.
2. A **BROWSER VIEW** is an external view that provides referential information about an object. At least one component of a browser view must be a menu. It might also include on-line documentation, or hand coded explanations of parts of an object. Such explanations might include pointers to other relevant objects, examples of how to use an object, and templates for creating instances of the object.
3. An **INSPECTOR VIEW** is an internal view of the private information of an object. It is used primarily for debugging, for example to examine the current state of an object or of the system.
4. An **ERROR HANDLING VIEW** is an external view. It is also used primarily for debugging, since it provides mechanisms for error recovery when defining a class and when sending or receiving inappropriate messages.

The simple conceptual model of objects, and views through which they can be manipulated, is accessible to both the programmer who first encounters Smalltalk and the more experienced user. Although the model is rather simple, it can be approached, modified and extended from a multitude of perspectives. The problem for both expert and novice is knowing how to get started on a project, and how to locate already existing objects. A Smalltalk environment can contain vast libraries of objects and information about them. A user who is not familiar with those objects may have a hard time finding the appropriate one. It is analogous to entering a library with a research topic. One must either possess good library skills, or have access to a competent research librarian. Smalltalk does not directly support either.

Smalltalk reversed the perspective on data and process. Boxer [diSessa & Abelson 86] goes a step further and changes the perspective on an object and the visual representation of that object. In most other programming languages, including Smalltalk, the object is an abstraction that has an associated display representation. One could almost think of the visual image of the object as a side effect of the object. Boxer takes the radical perspective that an object and its visual representation are one and the same.

Boxer is a recent experimental system intended primarily for "ordinary people" who would use a programming environment for mundane tasks such as correspondence, note taking, simple data analysis, and general "fooling

around". Boxer uses the *spatial metaphor* of a *box* to represent both data and processes in a very concrete way. Programming is simply the act of creating, examining the contents of, and otherwise manipulating boxes. Boxes are displayed on the screen and may contain text, graphics or other boxes. Text within a box can be used to create data structures, programs and comments. Nested boxes can be used for any sort of hierarchical structure; as a means for organizing a document such as a report or story; for organizing data such as in a telephone directory; and for proceduralizing a program, where nested boxes might represent internal variables and subprocedures.

The goal in building Boxer was to create an environment that allows "ordinary people to build personalized computational tools and easily modify tools they have gotten from others" ([diSessa & Abelson 86] page 859). The intended audience is different from most other programming environments. Traditional concerns such as formal simplicity, efficiency, verifiability and uniformity are not as important as more user-centered issues. These include:

- Understandability - the ease with which people can learn the system.
- Useful functionality - providing processes and data structures that may not be efficient or general, but are easily grasped.
- Simplicity of implementation - easily programming small simple tasks is more important than creating efficient complex ones.
- Fully integrated interactivity - user interfaces are not separable from the semantics of the language itself.

Boxer can also be characterized by *naive realism.* The computational world is no more nor less than what is seen on the screen. For example, consider a variable that is visible on the screen. If its value is changed either directly by the programmer or indirectly through a program, then the value that appears on the screen must change.

Since Boxer is a new system, it is too early to tell if users will be able to build and modify tools. These tasks will require that users learn the conceptual model of the system, and the skills needed to access existing tools. It is possible that Boxer will exhibit the same organizational and information access problems as Smalltalk.

2.3.4 Structure Editors Versus Viewing Systems

Structure editors and viewing systems provide two different perspectives on assisting the user. Structure editors provide a discipline that can be both supportive and restrictive. They prevent users from making syntactic errors, but also discourage them from developing personal data and program abstractions. Only the Gandalf Development System, which is for experts only, allows programmers to easily extend the support mechanisms of the environment to new computational abstractions. Viewing systems on the other hand are extremely flexible but merely provide the mechanisms for support and guidance. They encourage users to build their own support mechanisms. In this respect, both EMACS and the Interlisp facilities have more in common with viewing systems than structure editors.

The philosophical difference lies in the amount of flexibility and control to give the programmer. The trade-off is hard. Imposing a discipline supports a novice programmer, but may also inhibit an expert. Providing a simple, powerful abstraction in a concrete form may open the way for personal expression. It may also overwhelm a novice programmer who doesn't know where to begin. A possible compromise is discussed in section 5.

2.4 Goal Level Assistance

Both managerial assistance and non-textual interaction leave the task of constructing actual code in a programming language to the programmer. Neither has the *expertise* to construct the appropriate expression from a higher level specification. This aspect of consulting behavior would be helpful when the programmer wants to use standard techniques to solve a problem. Programmers would merely indicate the plan they wish to use. The system would then automatically generate code from the plan.

The Programmers Apprentice Project [Rich & Shrobe 78] is an ongoing research effort to develop systems that can offer such assistance. Its original intent was to explore programming on a level above the construction of semantically correct expressions[6]. Programming was seen as a skill that required knowledge intensive problem solving. The assumption is that an expert programmer satisfies a programming goal by sketching a plan and choosing standard algorithms and abstract data structures to carry out components of the plan. Those components are then coded in a particular programming language. The phases are not necessarily sequential. The choice of plans, algorithms and data structures are influenced by the nature of the language to be used.

Consulting behavior at this level must rely on knowledge of how to accomplish goals within that language. It must also include knowledge of programming goals that is independent of a particular language. Furthermore, it must include a vocabulary for communicating with the programmer about goals, plans and code. The Programmers Apprentice project introduced three notions that address these issues:

1. Assistance is provided by an *apprentice* who is not able to generate programs independently, but can assist a human expert. The assumption is that only experienced programmers will use the apprentice. The expert provides specifications for a program, and the apprentice generates the expressions in a particular language. The apprentice is *non-intrusive*, providing assistance only when requested to do so. The programmer is free to manipulate the apprentice's contribution at any level. The key to the interaction between the expert and apprentice is *shared knowledge*. Both are expected to posses a vocabulary that maps concepts such as "bubble sort," "simple report" and "user query" to expressions in a programming language.
2. Programming concepts are represented by a kind of template called a *cliche*. This is a nonpejorative

[6]Knowledge based programming environments [Smith *et al.* 85] present a different perspective, namely that the programming language itself support design level expression. An in depth analysis of this approach will not be presented here. The focus is on *very high level languages* rather than consulting behavior. Such systems encode goal level knowledge within the semantics of a formal language. The language includes mechanisms for specifying *transformations* from high level expressions to efficient, executable code. Transformations drive a refinement process that maintains a *correct* solution through all stages. Waters [Waters 85] suggests that the Programmers Apprentice may still be necessary within such systems. No matter how high the level of abstraction, programmers will still develop standard techniques for using these languages. The goal of the Apprentice is to assist in the implementation of these techniques.

description of a standard method for doing a task. Cliches embody the concepts that underlie the vocabulary through which the expert and apprentice communicate. Cliches differ from templates in structure editors in that cliches do more than provide a structure for a semantic expression. They provide a structure for the instantiation of a programming plan. For example, a cliche for a simple report on a data base might include the *roles* summarized in Figure 2. A cliche becomes a program through the detailed specification of roles, and the eventual encoding of that specification into an executable form.

3. The notion of a *plan formalism* is a knowledge structure that represents both the structure of particular programs and of cliches. Plans include knowledge of both data flow and program flow. The important distinction between plans and flow charts is that the former emphasize and explicitly encode information on the flow of data. The plan formalism provides a mechanism for accessing and combining cliches. It forms the backbone of how one can reason about, and find a solution to a programming goal.

A simple report on items in a database would include:

- **The title** that will be printed at the top of the report.
- **An enumerator** that lists some sequence of items.
- **Item information to be printed** that specifies what information should be printed about each item.
- **Column headings** that identify what is being printed about each item.
- **A summary** that might be printed at the end of the report.

Figure 2: Example of the Roles of a Cliche

In early work on the Programmers Apprentice, it was hoped that programmers would embrace the idea of plan specification [Waters 82]. Creating and editing programs would proceed through a series of plan refinements, where code was not actually generated until the end. However, experience with real programmers indicated that they were confused about how to use the planning vocabulary. They were more comfortable viewing and manipulating partially developed code throughout program development [Waters 86].

KBEmacs [Waters 85] grew out of the Programmers Apprentice Project to address this problem. It allows programmers to develop portions of Ada or Lisp code by referring to cliches and plans. It allows direct manipulation of code as if it were text. This puts a heavy burden on the system. It must be able to generate code that is equivalent to plan specifications, and also interpret textual changes in terms of plans and goals.

The current version of KBEmacs is a research prototype, and is consequently neither efficient nor robust. However, the limited experience with the system highlights an important point. Even expert programmers seem to need to view the development of a program through a single consistent representation of either LISP or Ada code.

2.5 Summary of Relief From Mundane Detail

Three kinds of relief from mundane detail were presented in this section. All attempt to remove the burden of *remembering* something. Managerial supervision removes the burden of remembering to *do* something. Non-textual interaction removes the burden of remembering how to *express* something. Plan level assistance removes the burden of remembering *how to do* something.

Two perspectives have influenced construction of the environments that contain this support. Structured programming encouraged the inclusion of restrictive mechanisms that guide programmers and prevent them from making mistakes. Exploratory programming encouraged the inclusion of mechanisms for extensibility and personalized styles. Both perspectives have influenced the design of environments for novice and expert programmers.

The only consideration so far has been how these mechanisms work. The assumption has been that the programmer knows about them, and knows how to use them effectively. However, the burden has merely shifted. Rather than remembering details, the programmer must now remember how to use these aids. This is no longer a matter of attending to detail. It is a matter of knowing how to make good use of a system. Consulting behavior that can be of assistance in this matter is discussed in the next section.

3 Providing Information: The Contribution of On-line Help

In order to work effectively in a programming environment, a programmer must know how to make efficient use of its features. Typically novice users learn a particular system by participating in a tutorial, reading an introductory text, or studying a manual. Some will have direct access to a human expert who not only has extensive knowledge about the system, but also has a knack for explaining things. This section is concerned with automated consultant behavior that is specifically directed toward providing information about the features of an environment in the form of *on-line help*. Three methods will be discussed:

- **Canned text with sophisticated access strategies** are becoming available in many commercially available environments. The help facilities of Digital's TOPS-20 operating systems [Tops-20 84] and Word Star's [Word-Star 80] word processor for microcomputers illustrate methods for using keyword and command languages. Three environments will be presented that have adapted techniques first developed in EMACS [Xinfo 84] and Smalltalk [Goldberg & Robson 83] for finding information by browsing through menus. [Borenstein 85] and [Magers 83] formalize some aspects of these systems and present empirical studies on their strengths and weaknesses.
- **Providing information that is tailored to the programmer's expertise** is primarily in experimental stages. [Kaczmarek & Sondheimer 83] present an integrated user interface that provides different methods of access depending on users' needs. [Rissland *et al.* 80] propose that users would be better informed by examples than by formal definitions. They have developed a taxonomy of example types.
- **Systems that provide information based on the programmer's goal** are also in developmental stages. [Finin 83] has built an *intrusive* system that notices when inefficient methods are used, and

suggests better alternatives. UC [Wilensky *et al.* 84] is a *nonintrusive* system that can answer natural language questions by inferring novice users' goals.

3.1 On-line Help That Relies on Canned Text

Current commercial environments rely primarily on canned text based systems for on-line assistance. Every command (function) that is part of the underlying system has an associated text. The text usually contains information on how that command can be used, what parameters it requires and what optional switches exist. The text might simply contain a formal definition of the command, or it might include examples of how the command can be used in certain situations. It might also include references or pointers to other relevant files. Access methods to canned text can be characterized as:

- *command languages*, in which the appropriate keyword or series of commands initiates a *help mode*,
- *menu driven structures* through which one can *browse* through a hierarchical structure of information on the system.

Most menu driven help systems contain an underlying command language. It is commonly presumed that more experienced users prefer command languages to menus, but that menus are essential for new users.

3.1.1 Command Language and Keyword Interfaces

Current systems that rely on canned text with a command language or keywords can be illustrated by the HELP feature of the Digital TOPS-20 operating system [Tops-20 84], and the prompt mechanism of the Word Star Word Processing Package for CPM-based microcomputers [Word-Star 80]. The major advantage of a command language or keyword is that the user has immediate access to a help file where pertinent information probably exists. The major disadvantage is that it is not always obvious what particular keyword is associated with a particular task. For the novice, the syntax of the command language may seem arbitrary and difficult to remember.

A command language interface may include command read-ahead, which allows the user to press keys that enable prompting of a particular command. For example, in TOPS-20, the command COP<esc> will be interpreted as a request to complete the command COPY, and a prompt will indicate what should follow. The system completes the command: COPY (file name). Another feature that is often present is a give-options key. After the user has begun a command, pressing the give-options key instructs the system to prompt with the possible alternatives. These features are extremely useful to both novice and expert users. Users can let the system "second guess" their intentions. The difference between these systems and DWIM is that these *remind* the user of the form of the command, but do not execute it.

3.1.2 Menu Driven Browsing

Current systems that rely on canned text with menu driven node browsing have been influenced by the Browser Views of Smalltalk [Goldberg 84][7]. It is important to note that browser views are *objects*, and are therefore manipulated with the same formal mechanisms as any other object in Smalltalk. The INFO system within EMACS [Xinfo 84; Stallman 81] provides similar features, although the underlying structure is text rather than object based. Adaptations of techniques developed in these systems are illustrated by the BROWSER system within PSL's NMODE [Browser 83], and the HELP system in the Apple Writer IIe word processing system [Apple Writer IIe 82].

The fundamental difference between INFO and NMODE is the textual organization. The canned text in INFO was specifically written for the hierarchical structure of the menu system, while NMODE simply accesses portions of the NMODE manual, acting as a sophisticated table of contents. The Apple system also simply provides an on-line table of contents. Due to the hardware limitations of the Apple II computer, the whole manual is not available on-line.

All of these systems offer a hierarchical organization of concepts that point toward pertinent canned text. To gain access to any particular concept, the user must proceed from a general to a more specific description of the task. In some systems, such as INFO, users are given canned text at each node, which may or may not encourage them to proceed further. In contrast, in Apple Writer the canned text is accessed only at terminal nodes, so that higher levels only aid in identifying which particular concept is sought.

Both INFO and NMODE also support a command language so that the more sophisticated user can bypass the menus. This flexibility seems to be an extremely important feature. Experienced users may fall back on the menus when the right command just won't come to mind. On the other hand even novices with little familiarity can get frustrated with the rigidity of a purely menu driven system such as that of Apple Writer. Referring to the manual is often a faster way to find information.

Unfortunately, even within INFO and NMODE, there is no way to immediately switch from command mode to menu mode and back. It is possible that a menu option might *remind* the user of more detailed information that can help aid the search, at which point menu browsing is no longer convenient. It is also possible that an attempt to find information through a command proves fruitless, at which point, expanding to a menu format could aid the search. The ability to do switching would require a more sophisticated user interface, one that keeps track of user queries, and relies on a representation of how to map directly from a menu node to a command and back again. Techniques

[7]The components of a browser view are described in section 2.3.3.

for developing such a system are discussed in section 3.2.1.

3.1.3 Studies That Evaluate The Effectiveness of On-Line Help

Two recent studies have evaluated the effectiveness of on-line help. Neither study was concerned with the development of smart systems using AI techniques. Both were concerned with the practical problem of including effective help in current commercial systems.

On-Line Enhancements Increase Usability

A study at Digital Equipment Corp. [Magers 83] indicated that superficial changes to the canned text and access methods would make on-line help more useful. The study looked at how computer novices were able to complete a series of simple file transfer tasks given minimal verbal instruction. A control group was given a standard DEC VAX/VMS operating system and paper manuals. The study group was given paper manuals, and a modified on-line help system. The method of generating explanations in the modified system was still limited to canned text but the nature of the interaction and the organization and content of the text were altered.

The control system canned text was primarily reference material that relied heavily on formal definitions. The modified system used a tutorial approach and was either jargon free or attempted to explain the jargon. The control system used mathematical notation to describe commands, the modified system used examples. The control text included lengthy explanations that filled more than one screen. The modified text was shortened into two-thirds of the screen sized frames. The control system text was oriented toward system commands, while the modified system was oriented toward user tasks.

The control system required the user to remember a keystroke sequence to get help. The modified system had a specially marked "help key." The control system was limited to a keyword indexed help structure. The modified system was context sensitive. The control system included rigid rules for forming HELP commands. The modified system was more lenient. While only error messages appeared in the control system when something went wrong, suggested corrections appeared in the modified system. The control system allowed the user to gain access to all of the system commands, while the modified system limited the available commands. The control system required the use of precise command names. The modified system included an on-line dictionary of synonyms.

The results of the study showed that the changes helped. Almost all of the users of the modified system (14/15) were able to complete the assigned task in under 60 minutes. Only three were able to complete it on the control system, and then at an average of 82 minutes. The help feature was used more frequently on the modified system, while users of the control system tended to rely on the manuals. Finally, users of the modified system showed

significantly more positive attitudes toward the computer in their answers to a post-experiment questionnaire.

The lesson to be learned from this study is that a system specification is not sufficient for providing information to users. It is not enough to simply put a manual on-line. Although expert programmers may be able to derive an explanation from a formal definition of a command, it may be more efficient in the long run to create canned text that is intended to *explain* how a command works. Furthermore, by making the interface more tolerant, all users, both novice and expert, will be encouraged to seek help on-line. The fundamental problem with the study was the number of variables that were introduced. It is not at all clear which of the many enhancements was truly helpful.

Quality Of Text Plays a Crucial Role

Borenstein's thesis work at CMU [Borenstein 85] indicates that while effective interactive facilities may be important, a crucial factor in the success of an on-line help facility is the quality of its text. His work explored the design and evaluation of on-line help. He was particularly interested in creating "good" support using current interface and data base retrieval technologies. He did not address the issues involved in developing sophisticated graphics or natural language interfaces or incorporating AI techniques.

Borenstein identified three components of a help system:

- The user interface, which includes the way help is requested and the way it is displayed.
- The structure of the data base that is searched to locate appropriate canned text.
- The text itself.

He built a prototype help system for UNIX called ACRONYM that attempted to use the best current technology for each component.

The ACRONYM interface consists of a 60 line display screen that was divided into three regions containing help text, a help menu, and a work area. Both the help text and the help menu are *context sensitive*. Their display is dependent upon the current context of the work area.

The system can automatically provide help by updating the help text and help menu areas whenever a command is entered in the work area. For example, if the user types "ls" and the space bar, a short text containing a description of how and what "ls" lists will appear in the help text region. Descriptions of other relevant information will appear in the menu region. The automatic updating facility can be turned off, and the user can also update the screens by typing "?" after any command sequence. The user can explicitly update the help regions by using the command "HELP" followed by the item about which help is sought. Finally, the menu screen can be browsed to locate other topics relevant to the current command.

ACRONYM's database is a network of objects that are *chunks* of indivisible text. The structure of the data base was intended to reflect the need to locate text from a multitude of different directions. Syntactic links are followed when text is sought following the parsing of a command line. Semantic links provide the facility for updating and browsing menus. The text in the database came primarily from a textbook on UNIX, and was therefore instructional and explanatory rather than definitional in nature.

Borenstein conducted an experiment in which both novice and expert UNIX users were asked to accomplish various tasks using one of four help facilities[8]:

- The standard "man/key" UNIX help facility, and text consisting of standard UNIX definitions.
- A hybrid system consisting of the man/key interface and the ACRONYM texts.
- The full ACRONYM system.
- A human consultant.

The success of the different facilities was statistically determined by how fast users could accomplish the assigned tasks. Among novice users the human consultant was the most effective, and the standard UNIX facility the least. The hybrid and ACRONYM systems fell almost exactly in the middle, with the ACRONYM system showing slightly more success. Among expert users, the human consultant was again the most successful, but not to the degree that it was among novices. Furthermore, the hybrid system was only slightly less successful than the human consultant. Finally, although the standard UNIX system was the least successful, the ACRONYM system was only slightly better.

From these results, Borenstein concluded that the most critical factor was the content of the text itself. Facilities that included the ACRONYM text were consistently better than those that didn't. The interactive component was less critical. Among novice users reading ACRONYM text, the facility that had a standard man/key interface was only slightly worse than the one with the sophisticated ACRONYM interface.

Among experts, the ACRONYM interface seemed to have been a hindrance rather than a help. Borenstein suggests that this occurred because experts already know how to use the standard man/key interface. They had to learn the new one, which may consequently have slowed their performance.

The experimental results do not allow conclusions to be drawn about the data base structure. Borenstein does

---

[8]A fifth experiment involving simulated natural language was conducted only on novice users. Users could type questions that were answered by a human expert hidden in another room. This facility was slightly less successful than those that included features of ACRONYM. This result is of questionable worth for two reasons. First the user had to *type out* questions which required more time than when simple pointing devices were used. Secondly, the menu in ACRONYM provided a *focus* for the user's attention and questions. Users of the simulated natural language facility were forced to ask questions without such visual prompting.

point out that the video tapes of the subjects indicates that users of the full ACRONYM facility seemed to engage in more *exploratory learning*. The menu and prompting facilities encouraged them to access information that didn't pertain directly to the task. This may have affected how ACRONYM was evaluated, since success was measured in terms of the speed with which users could finish tasks. Users who engaged in exploratory learning would naturally take longer than those who didn't.

3.1.4 Problems With Canned Text

Mager's and Borenstein's studies indicate that simple changes may be sufficient to provide useful generic on-line help. Both emphasize the need for quality text. Neither indicates what it should contain beyond that it be *instructional* rather than *definitional* in nature. Magers puts greater store in the quality of the interface than does Borenstein. Borenstein admits though, that an accurate measure of his interface may not be possible using his experimental methodology.

Borenstein's study also shows clearly that current help systems cannot compare with human consultants. Obviously there is something in the way that humans converse that is missing in current help systems. Sadly, Borenstein's thesis does not explicitly describe the interaction between the user and the consultant. Even without this information, four obvious differences emerge:

1. Oral communication is fast. The turn around time on asking and receiving an answer to a question will remain more efficient than any automated interface until human beings become more skilled at other modes of communication, and systems include interfaces that support true speech recognition and generation.
2. In natural language there are implicit rules for changing the focus of the conversation [Matthews 84; McKeown *et al.* 85]. Current interfaces are generally rather clumsy at changing focus. Frequently users must conceptually switch modes, figure out how to express the question they have, search for an answer, then figure out how to return to the task at hand. A question/answer dialogue between people is much more graceful.
3. In canned text help systems, the content of the text is generic, and will suit a spectrum of users. Natural language includes implicit rules for providing an answer that addresses the questioner's needs [Levinson 83; Paris 85a].
4. Finally, current help systems are unable to assist users within the context of their goals and plans. Current systems are only able to help with particular functions or features of a system. They are unable to provide direct help with a goal such as "deleting all my old files".

3.2 Taking Users' Needs Into Account

A common theme in section 2 was that novice and expert programmers require different kinds of relief from mundane detail. Novices need support that may be restrictive to experts. Experts ought to be able to extend a system without sacrificing basic bookkeeping details. It is likely that novices and experts also differ in the way they locate information. The fundamental distinction may be that experts know how and where to look and novices do not. The research described in this section attempts to meet the needs of a variety of users in a single system.

3.2.1 A Interface That Provides Alternatives

The Consul/CUE systems [Kaczmarek & Sondheimer 83] offer an example of how an integrated system of interface options can adjust to users' needs. More specifically, the system allows the user to make choices in how much control to give the system when asking for help. Those choices are made in a very natural way by taking advantage of a completely integrated, *mode-less* system. From the user's point of view, three different interface methods are simultaneously transparent. Furthermore, a request for help does not require "leaving the task mode," but is fully a part of the system.

CUE is a window and object based environment for interactive services that includes a command language, a pointing device and a menu interface. Integration is achieved through a large knowledge base of facts that formalize the environment. The knowledge base not only includes allowable operations on objects but information about how objects can be used, and what operations should result from specific user requests. Users not only execute instructions in a number of different ways, but request help using more than one format.

Users issue instructions to CUE using the command language or by selecting choices from the menu. Input that is neither a recognizable command or a menu selection is assumed to be a natural language request, and is passed on to Consul's experimental natural language interface. It parses a user request into a case-frame matching system, classifies it according to the knowledge base, and then maps it into a description that either invokes an operation or a request for help. It relies on information about the real world, the user view, and the allowable operations on system functions. It therefore "passes questions" back to CUE to get more information and proceeds to complete its analysis of the request.

The three modes of interaction are fully integrated. A user can switch from a natural language request to a command to a menu without overtly changing modes. For example, users might search through a menu for information, and decide that they don't know enough to make a choice. When interpreting a natural language request, the system takes into account users' position in the menu, and responds accordingly. Furthermore, users might give a command, receive an error message, then make the same request in natural language. Consul would attempt to match the query with the appropriate command and to fulfill the desired instruction. Although a specific request for help was not given, the system has indeed "helped" with the problem.

The Consul/CUE system is very appealing. Users may know some aspects of a system very well, while having little knowledge of some other part. The flexibility of moving from mode to mode allows users to gauge their interaction to their own level of competence.

3.2.2 Generating Examples That Address Users' Needs

Another study [Rissland *et al.* 80] addressed the problem of generating examples rather than using canned text in a help system. The work was based on the premise that pertinent examples are more powerful as learning tools than formal definitions. The more relevant the example is to the specific task at hand, the more likely it will answer users' questions. The assumption was that examples offer concrete illustrations but can also point out anomalies and counter examples. For instance, novices may use simple examples as *recipes*. Experts may use examples as templates to remind them of specific syntactic detail they may have forgotten.

[Rissland *et al.* 80] describes two systems, IA_LADYBUG and a subset of the VAX/VMS command language that incorporate a help system based on a taxonomy of example types. The help system relies on a pre-existing corpus of examples that are used as templates to customize responses. The system contains information on the user's level of expertise. It also knows how the pertinent concepts of the domain interrelate.

The taxonomy of example types consists of the following:

1. **Start-up examples** are those that involve easy perspicuous cases and relate most to what a rank novice would require.
2. **Reference examples** are the standard "textbook" examples that illustrate the basics of a concept.
3. **Model examples** are the paradigmatic, template-like examples that would refresh a user's memory.
4. **Counter-examples** are those that illustrate limiting or bad usages.
5. **Anomalous examples** illustrate ill-understood or strange cases.

The taxonomy allows for customization of the examples. Examples are generated by accessing an Examples Knowledge Base organized as a network of examples, text and procedures for modifying them depending on the context. Not only does the system give examples appropriate to the user's level of expertise, but also adjusts the examples to fit the present context. For example, the user might request information on why a particular command didn't work. The system might choose to generate a response using a "model example" that specifically includes references to the command and parameters originally entered by the user.

On two levels, the system takes users' needs into account in a very refreshing way. First, the content of the help has been carefully analyzed. The taxonomy of examples for a particular function requires careful attention to who will use a particular feature of a command and in what context. Most canned text methods fall severely short in this respect. Second, the organization of the knowledge base means that the system, rather than the user, has to do the work of locating the appropriate material.

3.2.3 Flexibility Is Not Enough

Both of the systems just described offer the user a large degree of flexibility. But at least two problems arise. The alternative interactive mechanisms in Consul/CUE let the user choose one of many paths to an answer. The problem still remains that the user must choose which path is best. The example taxonomy presents a different problem, since it assumes that the user has asked the right question. For example, the user may simply use the wrong vocabulary to ask a question, leading the system to make the wrong decision about what the answer should be. Although the example may be geared to the expertise of the user, it may not address the question.

These problems have to do with the user's intentions or goals. Although the knowledge base of both systems contains information that relates different types of helpful information to commands, neither representation contains information about users' goals. Consequently neither system is capable of reasoning about, much less providing a direct answer to a simple request such as "How do I .....?". The path chosen by users of Consul/CUE is determined by what they are trying to do. If the system knew about how specific commands relate to goals it might be able to generate an answer directly rather than making the user do the searching. In order to get an example from the Rissland et al. system the user must ask about a specific command. The user must draw the connection with the goal of doing something and the command that can do it. If the user asks about the wrong command, then no matter how informative the text, the goal still cannot be satisfied.

3.3 Systems that Attempt to Understand the User's Goal

In section 2 we saw how systems that contained surface level support led to research on how to support the programmer on a deeper level. Within the domain of help systems a similar progression has taken place. The help systems discussed so far do not contain mechanisms for determining what the user is trying to do. Without this crucial feature they are unable to:

- Notice when the user is doing something inefficiently and suggest an alternative way.
- Relieve the user of the burden of *searching* for the right information.
- Determine whether the user's request for help is actually relevant to the task.

Two systems will be discussed that explore how to automate these capabilities. Both are experimental artificial intelligence systems that are concerned with broader issues of natural language processing and problem solving. They are neither robust nor sufficiently fast for real users. Yet both illustrate that if a help system is to exhibit these behaviors it must possess knowledge and be able to reason about users' goals. Specifically, it must be able to make inferences about how particular features of a system can be used to accomplish tasks within the system.

3.3.1 Noticing Inefficient Methods

WIZARD [Finin 83] is an experimental system that provides information on a subset of the VAX/VMS operating system. Unlike help systems that wait for users to request assistance, WIZARD watches users' actions, infers their goal, and suggests a better plan when appropriate. WIZARD is primarily concerned with system efficiency. It is targeted for novice users who have learned the fundamentals of a system but may not make the best use of it. Finin points out that

> "it is quite common for a complex system such as a text editor or operating system to be designed so that a new user can learn a few basic commands which are sufficient to accomplish most tasks. Additional commands which greatly extend the convenience and practical power of the system are provided for the user to "grow into." It is common, however, for some users to become trapped by the simple complete set of basic commands and never progress to learning the full power of the system." [Finin 83]

Figure 3 provides an example taken from [Shrager & Finin 82]. WIZARD's response appears in italics. The user wants to change the name of a file and uses a plan that consists of two commands. WIZARD notices the plan and suggests a more efficient single command plan.

```
$COPY TEST.TXT EXP1.TEXT
$DELETE TEST.TXT

If you mean to be changing the name of
TEST.TXT to EXP1.TXT you might have
simply used the command:
     $RENAME TEST.TXT EXP1.TXT
The HELP command can tell you more
about RENAME.
```

Figure 3: Example of WIZARD Suggesting An Alternative Plan

WIZARD relies on a corpus of *bad plans* that are often used by novices. It attempts to locate a user's goal by matching the user's actions to those of the bad plans. Four factors must be considered when determining users' intentions from their actions:

- The actions that make up a plan may not be contiguous. For example, a user might begin a task, put it aside, then resume it later.
- A single command may play a role in more than one plan.
- A sequence of events may map to many plans that may map to many different goals. There is a large degree of *ambiguity* in determining what plan is being executed by a sequence of actions, or what goal is being satisfied by a particular plan. Conversely a plan may be instantiated by numerous sequences of actions, and a goal may be satisfied by one of a number of plans.
- Extensive knowledge of the user's environment is necessary to determine possibly dangerous side-effects.

WIZARD represents both the abstract concepts of the system and the actions of a particular user. A demon is activated when an action initiated by a user matches the initial event of a bad plan. The demon monitors subsequent actions for further development of that plan. If the entire plan is instantiated, WIZARD begins generating advice using an *advice template*. Slots in the template are filled by the context of the current interaction.

WIZARD is primarily concerned with determining whether a user's actions constitute a bad plan. It relies on its ability to monitor and synthesize those actions. It was not designed to answer explicit user queries, nor does it seem likely that it could generate a *good plan* for a user from scratch.

WIZARD's ability to detect inefficient actions illustrates an important aspect of consultant behavior. Yet it fails to consider the user's receptiveness to its advice. WIZARD only presents advice if similar advice has not been given before. This may not be the most useful perspective, especially in a system that cannot be explicitly queried. Consider a user who does something inefficiently. WIZARD gives some advice, but for whatever reason, the user chooses to ignore it. Later, in a similar situation, the user attempts to execute a set of actions, remembers that there is a better way, but can't remember what it is. Since the user cannot directly query WIZARD, the better plan can not be retrieved.

3.3.2 Understanding Novice User's Questions By Inferring Goals

UC [Wilensky *et al.* 84], a consultant system for UNIX, is an experimental system that attempts to answer novice users' questions about UNIX. The primary intent of UC is to explore issues of natural language processing and planning. The domain of UNIX consulting was chosen because it is rich enough to provide interesting problems in communication and sufficiently restricted to bound the knowledge required by the system.

The UC project addresses basic problems of novices. Often, beginning users do not know the name of the command that will accomplish a task. Most help systems that are organized in terms of those commands offer little real help. The access methods often require users to draw the connection between the task and the proper command themselves. When the wrong command is chosen, the help system is merely used to search for an explanation. Most novices do not possess the skills to use these facilities effectively.

UC helps novices by mapping tasks to commands for them. It allows users to converse in the familiar medium of the English language. UC can answer questions about:

- how to accomplish typical UNIX tasks such as creating, moving and deleting files,
- the names of commands, and the functions of their switches,
- definitions of operating systems terminology, including UNIX specifics,
- why commands may not have produced the expected results.

UC is a "complete" natural language interface to a data base of information on UNIX. As such, it contains:

- A natural language analyzer call PHRAN (PHRasal Analyzer) that reads sentences and produces a conceptual representation of their meaning. PHRAN uses a pattern matching mechanism to take portions of English sentences and match them to a data base of pattern-concept pairs. As a language understander, it is sufficiently robust to rarely exhibit "hard" failures. It includes a context model so that it can begin to handle certain pragmatic concerns such as discourse focus, reference and disambiguation.
- A goal analyzer that takes the conceptual representation constructed by PHRAN and attempts to interpret the task specific goal of the user. Presumably, the concept produced by PHRAN is a request for help, but the intent may not be explicitly stated. For example, the user may type "I want to delete a file". The system must *infer* that the goal of the user is to receive information on how to delete a file. The goal analyzer uses a planning mechanism based on earlier work by Wilensky [Wilensky 83]. It uses a set of rules that consist of an input and output pair of conceptualizations related to possible goals. If the PHRAN generated concept matches the input component of a pair, then the output component is inferred. If an output component cannot match the input component of another pair, then it is presumed to be the goal of the user.
- A plan generation mechanism called PANDORA that takes the inferred user goal and attempts to create a plan that will satisfy the goal. For the most part, PANDORA simply locates a plan stored in memory that directly satisfies the goal. PANDORA does have more sophisticated planning strategies involving *meta-planning*. For example, it is able to detect when a particular plan may interfere with an implicit goal. Consider the plan "delete all files" for the request "I need more disk space". Although the plan satisfies the goal, it interferes with a general goal of "preserve useful files".
- A natural language generator called PHRED (PHRasal English Diction) that produces English text from the representation of the plan produced by PANDORA.
- A User Modeling Component called KNOME (Knowledge Model of Expertise) [Chin 86] that uses a strategy of *double stereotyping*. It has knowledge about typical user expertise and a classification of commands by "level of difficulty." By doing an analysis of the content of the user's question in relation to the user's expertise, KNOME generates inference rules that can be used by all of the other components of UC. For example, KNOME can deduce a "level of expertise" of a user from the initial question. It can use that deduction to determine the user's goal. Presumably novices have more simple goals than experts. Finally, it will influence the plan produced by PANDORA, and the text generated by PHRED, on the assumption that different user types require different kinds of explanations.

Although the developers of UC hope to produce a robust system within the next decade, the primary emphasis of the project is on broader issues of natural language processing. Its major contribution to help systems is articulating and demonstrating the importance of inferring users' goals. By doing so, a system that attempts to provide information can be of more immediate help to a novice user.

UC is not directly concerned with how to provide goal-oriented help to non-novices. Presumably more expert users would seek information that does not merely describe how a command accomplishes a particular task. They might be concerned with clarifying what a command does, or how parameters can be adjusted. As Finin suggests, more expert users might also be interested in finding more "efficient" ways of doing things. Although UC ought to be able to provide answers to these kinds of questions, it doesn't seem to possess the knowledge to do so. Furthermore, the computational energy required to parse the user request, develop a plan, and generate an answer seems exorbitant.

3.3.3 Problems with Inferring Users' Goals

Both UC and WIZARD assume that even novice users know what they want to do with the system. This assumption is suspect because novice users may only have a vague or ill-defined idea of what they want to do. Even if they have a clear idea of the task, they may not know how to execute it within a particular system, or how to begin to ask questions about it.

WIZARD infers users' goals from their actions, and assumes that those actions indicate a particular goal. UC assumes a goal not only from users' actions, but from their questions. But if the goal is ill-defined, or if the actions do not suit a particular goal, then the help provided may not be appropriate. [Pollack 86] articulates and suggests solutions to this problem. She describes the problem as a discrepancy between the beliefs of the questioner and respondent regarding actions in the domain. In order to be informative, the respondent must take this into account while analyzing the user's plan, and generating an answer. Pollock's prototype system SPIRIT demonstrates a plan inferencing mechanism that includes rules for dealing with conflicts between beliefs[9].

3.4 Summary of Providing Information

This section has discussed how help systems provide information to users. Two concerns have been how users can ask for help, and what is presented as a response. All current commercial systems use either a command language or a menu system to access canned text. These systems are unable to adjust to users' growing expertise. Experimental systems address this by providing alternative access mechanisms and generating text that depends on users' knowledge and needs. Finally, we discussed how users' plans must be taken into account, and described systems that deduced users' plans from their actions and the questions they asked.

Yet all of the systems fail to address three important issues:

1. They provide information on how to use the programming environment, but not on how to get help.
2. The information they provide is restricted to individual commands of the system.
3. The information they provide does not consider *why* the user asked for help. They do not distinguish between reminding the user about something, and introducing something new.

All of the methods and systems described assume the user has been introduced to the fundamentals of getting information. They also assume that the information is primarily intended for novices. In the systems that relied on access methods and canned text, we must assume that some facility exists for learning how to use the access method. The assumption in natural language systems is that the user will know how to type questions in English. But as Pollack suggests, users may not ask the right question regardless of whether they are using a command or natural

---

[9]It would be inappropriate to discuss her work in depth here, as a careful analysis would take us too far afield into computational linguistics. Unlike [Wilensky *et al.* 84] she has not attempted to build a robust consultant system.

language. Help systems ought to be able to help novice users learn how to get help. Furthermore, it cannot be assumed that experts do not need information. It is more likely that they simply need it in a different form. In fact most of the command language/canned text systems are more useful to experts than to novices.

Part of what differentiates an expert from a novice programmer is the number of commands they know. But another important distinction is the *techniques* that each uses. Such techniques are usually not just simple commands, but *sequences of commands* that accomplish a task. Experts go so far as to formalize these techniques in batch files or macros. One can think of them as as descriptions of plans that accomplish very high level goals. Finin's work begins to address this. He is only concerned with providing alternative, more efficient single commands or concepts, not necessarily sequences of them. Wilensky also suggests that plans might be accomplished by sequences of commands, but provides no immediate solution.

Finally, none of the systems present information in a form that directly considers *why* users asked for help. The concern here is not with the access mechanisms, but with the content of the text. A distinction is often made between definitional and instructional text, yet only the Rissland et al.'s work on an example taxonomy and Chin's user modeling stereotypes in UC began to articulate what might distinguish these styles. We propose that definitional text is more appropriate when the user needs to be *reminded* about something. Instructional text is more appropriate when the user needs to learn something new. The issues involved in generating the latter will be explored in the next section.

4 Providing Instructional Information

In the previous section we saw how the problem of providing help can be split between the access mechanisms and content of the information a system provides. This section is concerned with how work in Artificial Intelligence has approached the problem of providing instructional information. This work is only tangentially concerned with consulting behavior in programming environments. Its primary focus has been to understand the complex interactions of tutors and students.

Tutoring is relevant here because a critical part of consulting is knowing when to tutor and when to simply remind or inform. A user who forgot the syntax of a command does not want to wade through a tutorial. On the other hand, a user who is looking for a new command may need an instructional rather than definitional approach. In order to understand the complexity of choosing between informing and tutoring, it is necessary to survey systems that tutor.

Research on Intelligent Tutoring Systems (ITS) has contributed to theories in many areas of AI including

natural language processing, planning and problem solving, and knowledge representation [Sleeman & Brown 82]. A full discussion of ITS research would not be appropriate here. This section presents work that pertains directly to tutoring procedural skills in a non-curricular manner. Learning techniques and commands in a programming environment is procedural rather than factual. One learns *how to use a command or technique*, not what either *is*. The tutoring situation is mitigated by the current task of the user, not by a prescribed sequence of activities pre-determined by the tutor.

Three important issues must be considered when tutoring procedural skills:

- **Diagnosing user misconceptions** allows the tutor to focus directly on the students' needs[10]. BUGGY and DEBUGGY [Brown and Burton 78; Burton 82] demonstrate that this not an easy task. [Sleeman 82] stresses the need for separating diagnosis from tutoring, and offers a production system-based model.
- **The strategy used to engage the student** is influenced by the environment. Two possible approaches are *coaching* and *guiding*. [Goldstein 82; Burton & Brown 82] provide examples of coaching in exploratory environments. [Miller 82; Anderson 86] provide examples of guiding in more structured settings.
- **Diagnosing complex skills requires understanding students' plans.** [Genesereth 82; Johnson and Soloway 83] show that tutoring in domains such as simplifying mathematical expressions and programming can be improved by analyzing students' underlying goals.

4.1 The Importance of Understanding Misconceptions

A simplistic assumption about tutoring is that information need only be presented once. People, unlike machines, often miss crucial parts of instructions or do not pay attention in the first place. Consequently, tutoring and, by extension, consulting, involves more than presenting a body of information. It must include the ability to notice and remediate misconceptions and misunderstandings on the part of the student.

The complexity of this problem in computer tutors was first articulated by [Brown and Burton 78] in their systematic study of errors in simple arithmetic[11]. Their work began during a progressive movement in math education [Solomon 86]. It was commonly assumed that rote memory of math facts was insufficient for developing lasting understanding of mathematics in children. Mathematics education, starting with simple arithmetic, must include teaching children *how to think about* as well as *do* mathematics [Papert 72]. When learning simple "math skills" such as subtraction, it was insufficient to simply drill children until they "got the right answers." It was

---

[10]Related work [Mays 80; Kaplan 82; McCoy 83] from computational linguistics will not be discussed here, since it is primarily concerned with factual rather that procedural knowledge. Furthermore, a comprehensive discussion would require an analysis of issues that pertain more to natural language question answering than to consulting behavior. For a discussion of this work see [Paris 85b].

[11]Brown and Burton began looking at tutoring procedural skills in the more complex domain of electronic trouble shooting through the SOPHIE projects [Brown *et al.* 82]. The domain proved to be extremely complex, both in terms of developing a "reasoning engine" that could understand problems, and a "tutoring system" that could provide appropriate instruction. Building a reasoning engine for subtraction was much more straightforward, and therefore allowed them to concentrate on tutoring.

important to see *why* they made mistakes, that is, *how their thinking was buggy*.

4.1.1 Diagnosing Bugs In Simple Arithmetic

Burton, Brown and their collaborators focused on the problem of trying to understand why students made mistakes on simple arithmetic problems. They hoped to show that errors were neither the result of cognitive lapses such as fatigue and laziness nor totally random in nature. They proposed that students errors manifest "conceptual bugs" in the mental procedures they use to solve arithmetic problems. If this were the case, then simple drill and practice would be inappropriate for remediation. Instead, explicit "debugging" should take place by tutoring the student on the particular misconception.

Consider a student who, given a series of subtraction problems, provides the answers shown in Figure 4. Clearly, the student doesn't fully understand subtraction with borrowing when "0s" are involved[12] . A simple remedial approach would present the correct solutions, and possibly "walk through" the procedure as in Figure 5. This type of explanation seems to assume that the student will simply abandon whatever strategy was used to produce the incorrect solution and adopt the procedure just presented. There is psychological evidence that children and possibly adults do not do this easily [Groen 79].

```
1)    600          2)    413
    -  53              - 137
      ---                ---
      653                336
```

Figure 4: An Example Of A Systematic Error in Subtraction

Implicit in the assumption is that students will "learn from their mistakes" and notice what they did wrong. In the case of the second problem, subtracting 3 from 10 in the ten's column should result in 7, not 3. *If* students paid attention to the explanation, and were sufficiently perceptive, they might compare their faulty method with the correct one in Figure 5, and realize why their method was inappropriate. One would expect the tutor to guide, or at the very least, participate in such an analysis. Yet in order to do so, the tutor must have extensive knowledge not only of the domain but of typical mistakes made by beginners in the domain.

This is precisely the problem addressed by Brown and Burton. They began by developing a corpus of wrong answers to simple subtraction problems. They discovered that certain types of errors occurred systematically and were able to explain them in terms of misapplied or faulty steps in procedures that satisfy simple computational

[12]The first problem indicates that the student doesn't know how to "borrow from zero". The second problem shows that the student does know how to "borrow from digits other than zero" to the point of keeping track of the "zero left over" in the ten's column of the second problem. The problems and solutions were contrived by us based on known bugs from the corpus of Brown and Burton.

```
In order to subtract 7 from 3 you must
borrow from the ten's column.
Then 13 - 7 = 6, which is the result in the one's column.

Since you borrowed, the problem in the ten's
column is to subtract 3 from 0, so you must
borrow from the hundred's column.  10 - 3 = 7,
which is the result in the ten's column.

Since you borrowed, the problem in the hundred's
column is to subtract 1 from 3.  So the answer
is 2 in the hundred's column.
```

Figure 5: Simple Remediation on Problem 2

goals. Their analysis and subsequent diagnostic systems relied on a "procedural network model" that was organized around simple goals in subtraction. A goal might include numerous alternative correct and incorrect methods for achieving that goal, appropriate conditions for satisfaction, and diagnostic guides. They intentionally chose a very "loose" representation because:

> We did not know what primitives or control structures would be appropriate so, in an attempt to avoid having our language shape the way we saw the data, we chose a representation language which allowed us freedom to capture each newly observed set of student errors in a way that seemed most appropriate to those errors. Thus our initial representation scheme was *ad hoc by design*. ([Burton 82] page 159)

The process of "doing a subtraction problem" either correctly or incorrectly could be traced by following a path through the network. Two constraints restricted the number of unique paths. First it was assumed that bugs occurred as "the least possible variant of a correct skill" ([Burton 82] page 159). The path that describes an incorrect solution branches to faulty methods as late as possible. Secondly, it was acknowledged that compound bugs do occur, and these must be represented as a composition of their component parts. In all, 110 primitive and 20 compound bugs were identified and incorporated into the network.

It must be stressed that diagnosing the cause of a student's misconception is not as simple as tracing a path for a single subtraction problem through the network. Various kinds of noise such as fatigue and inattention can interfere with pinpointing conceptual rather than superficial bugs. In order to infer that a student has a conceptual bug, errors must occur *consistently*. Furthermore, bugs may interfere with each other, producing a correct answer. The diagnostic systems DEBUGGY and IDEBUGGY took this into account.

DEBUGGY is a non-interactive system that analyses student answers on a carefully constructed "off-line" paper and pencil test. It uses sophisticated domain-dependent heuristics that take noise and consistency into account. IDEBUGGY is an interactive system that also presents carefully chosen problems, but refines its hypothesis after each problem. Therefore later problems are automatically generated to test earlier hypothesizes.

Both systems are reasonably successful in diagnosing real bugs in real children.

4.1.2 Explicit Representation of Misconceptions in More Complex Domains

BUGGY and DEBUGGY showed that diagnosing misconceptions in a task as simple as subtraction is extremely knowledge intensive. Similar conclusions have been reached in more complex domains such as algebra [Sleeman 82], naive meteorology [Stevens *et al.* 82], and medical diagnosis [Clancey 82]. Although both meteorology and medical diagnosis require understanding causal events, neither really involves mastering a procedural skill. Both do require "procedural reasoning", but the objects that one manipulates are thoughts rather than things. Consequently, an in-depth discussion of manifestations of buggy thinking requires natural language analysis that will take us too far afield. The precise language of algebra and subtraction do not present such problems.

The Leeds Modeling System (LMS) [Sleeman 82] uses a production system to build a model of students' understanding of algebra. The system contains rules for manipulating algebraic expressions and "mal-rules" that describe typical student errors. Figure 6 illustrates a rule and mal-rule related to solving for a variable on the left hand side of the equation. In the correct rule the sign of M is changed as it moves from the left to the right hand side. In the mal-rule its sign is incorrectly left the same. In each case the rule identifies how a pattern results in an action. The system builds a model of the student's knowledge by noticing when rules and mal-rules are applied.

```
RULE  applied to PATTERN results in ACTION
(rule)
NTORHS          (lhs +/- M = rhs)   (lhs = rhs -/+ M)

(mal-rule)
MNTORHS         (lhs +/- M = rhs)   (lhs = rhs +/- M)
```

Figure 6: A Rule and Mal-Rule For Solving an Algebra Problem[13].

4.1.3 A Theory For Why Misconceptions Occur

LMS and the DEBUGGY systems isolate common misconceptions in terms of bug types. They reduce the complexity of building a model of the student's understanding. Although the diagnosis process in both can identify a conceptual bug, neither cannot identify *why* the student developed it in the first place. Two theories that make an attempt at an explanation will be presented here. A third [Soloway & Ehrlich 84] that focuses on learning to program will be discussed in section 4.3.2.

[Brown & VanLehn 80] present a principled theory called "Step and Repair Theory" based on the BUGGY

[13]Taken from [Sleeman 82]

work. They suggest that learning a complex skill first involves developing procedures that contain the proper steps. Someone with minimal skill has a limited number of procedures. When confronted with a task, one attempts to choose the correct procedure. If a correct procedure cannot be found then one attempts to "repair" or modify a known procedure. Bad repairs can cause errors, and can be viewed as conceptual bugs. They occur when subcomponents of a skill are not sufficiently mastered.

[Matz 82] proposed a similar theory to account for errors caused by students in high school algebra. Her work describes how typical errors in manipulating algebraic expressions can be explained in terms of misapplication of known procedures. She differentiates between incorrectly applying known rules *as is* or *adapting* them to new problems.

[VanLehn 83] argues that one cannot simply hope that properly structured teaching will eliminate bad repairs. A procedure or explanation that seems abundantly clear to even the best of teachers may be utterly confusing to the most eager student. He claims that before we can hope to build truly robust tutors we must know more about how people learn. Specifically we must develop a better understanding of how people take discrete pieces of information and combine them into complex and sometimes faulty knowledge structures. We will return to this problem in 4.3.2.

Van Lehn's claim should not be seen as a suggestion to abandon the attempt to build computer tutors. But like other areas of Artificial Intelligence, researchers must be careful about the task they set out to accomplish. A distinction must be drawn between practical systems that only mimic real intelligence, and experimental systems that test hypothesizes. Practical systems such as the LISP Tutor which will be discussed in section 4.2.2 are currently available, but tend to over-simplify the tutoring process by presenting a single perspective. Exploratory systems are still being developed. The claims made about such systems must be carefully considered in the context of the educational theory they hope to advance. In either case, evaluations of these systems must be tempered by how little anyone really knows about the "domain" of learning and teaching.

4.2 Systems that Tutor New Skills

Most of the work on misconceptions was not concerned with how students were initially introduced to the subject matter. Two approaches, computer as coach and computer as guide, will be discussed here. A coach is someone who watches performance and occasionally introduces better strategies. A guide is someone who carefully structures an activity to communicate a specific set of ideas or skills.

4.2.1 Computer As Coach

WUSOR [Goldstein 82] and WEST [Burton & Brown 82] were experimental systems that attempted to study opportunistic tutoring. Like BUGGY they were built in the 1970's during the time when exploratory computer learning environments for children were being developed. The intent of such environments was to provide a rich "microworld" in which students can use their natural abilities to solve problems. In the process of playing a game or getting the computer to do something for them they develop skills at their own pace and for their own purposes [Papert 80][14]. The problem with such environments is that opportunities for learning something new are often "felicitous" [VanLehn 83]. Learning may or may not occur when a "coach" suggests a better strategy after noticing a deficiency in the students thinking. Both WUSOR and WEST were developed for coaching game playing that required math and logic skills. Both illustrate the importance of diagnostic ability and communication skills to coaching.

WUSOR Relies on a Genetic Graph

WUSOR [Goldstein 82] illustrates the complexity of modeling a student's development in an exploratory environment. In order to provide the right suggestion at the right time, the coach must know how to exploit what students know to teach them something new. WUSOR coaches a game called WUMPUS in which a player explores a geometric lattice of caves. A WUMPUS lives in the caves along with other dangers such as bats and pits. The object of the game is to shoot the Wumpus before it finds you, while avoiding the other dangers. Dangers in nearby caves can be detected and provide clues as to where to move next. Simple rules of logic and probability govern good moves. The pedagogical objective of the game is to develop those rules. WUSOR "watches over a player's shoulder." When a player's move differs from the one WUSOR considers "best", WUSOR suggests the better move to the player.

The first version of WUSOR was unable to characterize the relative difficulty of various rules and presented complicated suggestions that overwhelmed novice players. A second version divided the set of rules into five skill levels. In this version, a rule was not included in a suggestion until WUSOR "believed" the student was familiar with the preceding levels. But simple skill levels were also found to be insufficient. Learners do not simply progress through tidy stages, but build new knowledge from old using many different approaches.

The latest version of WUSOR uses a "genetic graph" to encode the evolutionary relationship between skills. These relationships reflect learning by analogy, refinement, correction and generalization. The nodes of the graph

[14]A full discussion of the idea of educational computing environments, how they are implemented and their practical applicability will once again take us too far afield. [Solomon 86] provides an excellent summary and analysis.

are the logical and probabilistic rules that expert WUMPUS hunters take for granted. The links encode the relationship between rules. By following unique paths through the Genetic Graph, WUSOR can generate a suggestion that more accurately reflects the varying backgrounds of different players.

WEST Uses Heuristics To Choose When To Coach

WEST [Burton & Brown 82] was developed to explore how and when to offer advice. WEST coaches a computer game called "How the West Was Won" that drills simple arithmetic facts. Players move a number of steps on a "board" by combining the single digit results of three "spinners" with simple arithmetic operators. No operator may be used more than once in any move. The object of the game is to be the first player to land exactly on "home". Short cuts and "safe" locations on the board reduce the desirability of always choosing the largest possible result. Since the pertinent rules and strategies are rather straightforward, diagnosis is less of an issue than some other domains. Of more interest here are the tutoring heuristics that were articulated by this work. The heuristics are intended to let the player engage in productive exploration, even if it involves making mistakes. The coach only interferes when a player seems to "get stuck" on a particular idea such as "always use the maximum values."

In WEST rules and strategies of playing the game are organized in terms of *issues*. These include knowledge about the basic arithmetic operators, about strategies such as using a short cut or landing on a safe location, and about general game-playing strategies such as watching what your opponent does. WEST builds a model of students by comparing their behavior with that of an idealized expert. It identifies significant discrepancies as *weaknesses*. When students choose a move, WEST compares the issues involved with any of the students' weaknesses. It then uses the heuristics to determine whether and how to coach. Coaching is done by providing *example* moves that illustrate the issue. The coaching heuristics summarized in Figure 7 come from [Burton & Brown 82].

WUSOR shows the complexity of choosing *what* to coach in an open-ended environment. WEST articulates the complexity of determining *when* to coach. One might assume that if good coaching is so difficult, why not simply guide a student through a carefully planned sequence of instructional material. The disadvantages of this approach will be discussed in the next section.

4.2.2 Tutors That Guide

A simple approach to introducing new information or skills is to "guide" a student through an example session. Determining what the student knows and what to introduce and when is not nearly as complicated as in open environments in which coaching takes place. In AI terms, since the sequence of events that lead to learning is controlled by the guide, the search space of both what and when to tutor is severely restricted. Researchers with a

- Pedagogical:
  1. Make sure player is weak in the ISSUE before giving advice.
  2. Only use an EXAMPLE if the alternative move is dramatically superior.
  3. After providing an EXAMPLE, let the player redo the move, affording the opportunity to immediately apply the ISSUE.
  4. If a player is about to lose, don't tutor unless tutoring will help win the game.
- Maintain Player Interest:
  1. Under no circumstances tutor two consecutive moves.
  2. Allow players time to discover ISSUES on their own.
  3. Comment on exceptionally good moves, explaining why they are good, don't just criticize.
- Increase Chances of Learning
  1. If the player is playing against the computer, not another person, have the computer play an optimal game.
  2. When players ask for help, provide hints that address their weakness, describe the possible moves, and explain why the optimal move is best.
  3. If the player makes what seems to be a careless error, provide explicit commentary just in case it wasn't.

Figure 7: Summary of Heuristics Used By WEST

Piagetian perspective argue that the learning that takes place is not as significant as in coached environments [Groen 79]. The student may not have much motivation for participating in the activity. On the other hand, if the student is motivated, guiding a student through a tutorial dialogue is certainly more efficient. The system does not have to wait for an opportune moment to introduce an idea, it creates the moment.

Two systems will be discussed here that focus on introducing and re-enforcing a particular perspective. [Miller 82] describes an early attempt to encourage systematic planning skills in learning to program in Logo. [Anderson 86] discusses current work on introducing basic concepts of LISP. Both systems use a highly structured format to guide a student through the learning process.

SPADE - A programming tutor that encourages planning

SPADE was an experimental system that investigated how novice programmers could be encouraged to develop planning skills. It explored

> the hypothesis that articulating one's problem-solving strategies facilitates learning, by providing a vocabulary of concepts for describing plans, bugs and debugging techniques ([Miller 82] page 119).

SPADE was a highly structured interactive environment in which one could articulate plans for writing programs

that used Logo turtle graphics[15]. SPADE relied on a simple taxonomy of problem solving behavior. One can *identify* a known technique or primitive, *decompose* the problem into subproblems, or *reformulate* the problem into a more manageable form. For example, one can decompose a problem by noticing that the solution requires repeating a subproblem.

Tutoring in the first version of SPADE consisted of prompting students to explicitly choose methods from the taxonomy in order to solve an assigned problem. The student did not express a plan directly in Logo code, but used a "refinement" language to develop the program. SPADE would engage the student in a dialogue that encouraged top-down program development. First the goal of the program would be identified. Then SPADE might ask the student whether the problem can be reformulated, or decomposed. If the student chose to decompose it, then SPADE would ask the student to articulate the component parts.

SPADE performed routine bookkeeping tasks such as maintaining information about the degree to which components of the plan were implemented. SPADE could therefore engage in discussions about testing and debugging that pertained to unimplemented plan components. Students were encouraged to discuss the problem by referring to parts of the plan rather than directly to the code. SPADE had very limited interactive facilities. It relied heavily on canned text and simple multiple choice menus. Furthermore it was only able to help students plan solutions to a few very simple Logo problems.

The first version of SPADE only knew how to manipulate plans. It did not know how to translate plans into specific code. For example, it could notice that an initial step in a decomposition had not been implemented, but could not participate in a conversation about how it should be implemented. A user might specify that a component of a problem was to draw a roof and that the roof would be drawn with a triangle. Although SPADE understood the progression from problem component to triangle, it could not understand what a triangle was.

A second version of SPADE incorporated knowledge necessary for this kind of problem specification. In order to prevent it from simply dragging a student through a predetermined solution, it was modified to exhibit some coaching behavior. The second version of SPADE also ran into many of the implementation issues discussed in section 2.4 regarding the Programmers Apprentice.

Anderson's LISP Tutor Is A Practical Guide

[15]Turtle graphics is an alternative to cartesian geometry which is particularly well suited to computational exploration [Abelson & diSessa 81]. By providing parameters to the four basic movement commands, FORWARD, BACK, RIGHT and LEFT, one can move a "turtle" and draw pictures that embody geometric principles.

The LISP Tutor and the PUPS Tutoring Architecture [Anderson 86] are examples of systems that trade flexibility for efficiency and usefulness. The LISP Tutor is a robust system that has been used extensively to support an introductory course in LISP at Carnegie-Mellon University. The PUPS Tutoring Architecture is intended to generalize the mechanisms of the LISP tutor and is still under development. Both take advantage of a highly restrictive environment to narrow the range of potential tutoring dialogues.

The LISP Tutor presents a student with typical introductory problems that illustrate basic LISP techniques. For example, computing the factorial of a number might be used to illustrate recursion. The Tutor monitors every step of the student's attempt to code a solution in LISP, assisting in planning, checking for syntactic errors, and intervening when the student drifts away from the correct solution. When the student displays typical misconceptions, the Tutor initiates remedial sessions that take the student away from the coding problem and review basic concepts. For example, a student who has trouble writing the recursive step of the factorial function might be asked to compute on paper the factorial of a decreasing sequence of numbers. The system would ask the student to notice the relationship between the results and choose a multiple choice answer that best articulates the relationship.

The LISP tutor uses a production system to represent simple programming goals, and associated correct and buggy actions. The system contains approximately 1200 productions. Figure 8 shows two of these productions; a correct and buggy rule for coding a test for zero. Correct and buggy solutions to the tutored problems are generated by running the problems through the production system. The resulting traces simulate typical correct and incorrect student behavior. Tutoring dialogues are attached to points in the trace where the simulated behavior indicates a misunderstanding on the part of the student. The Tutor does not intervene as long as the student exhibits behavior that follows the *correct* path. When the student's actions match those of a *buggy* path, the Tutor initiates a dialogue by providing some canned text commentary. The Tutor elicits responses from the student through multiple choice questions.

The LISP tutor suffers from four problems:

1. It requires that coding always occur in a top-down fashion. Although this encourages a generally accepted programming style, it is not always appropriate for beginners who may not have a clear idea of how to tackle a problem. This problem will be discussed more fully in 4.3.2.
2. It is even more restrictive since it requires that coding develop in a strictly left to right manner. For example, initializations of variables in a loop must be encoded before the loop itself.
3. Since the Tutor monitors every symbol typed by the student, it is highly intrusive. For example, a simple typing error might initiate a tutoring session at the same time that the student notices it, and attempts to press the delete key.
4. The mechanisms for engaging in a dialogue are extremely primitive. The canned text and multiple choice based interactivity are restrictive and at times awkward. For example since the student's input to the dialogue is restricted to responses to multiple choice questions, the student can participate in the

A "Correct" rule for checking for a value of zero.

```
IF   the goal is to test if a value
     is equal to zero
THEN call the function ZEROP and set
     a subgoal to code the value
     to be given as an argument to the function
```

A "Buggy" rule for checking for a value of zero.

```
IF   the goal is to test if a value
     is equal to zero
THEN use the function EQUAL and set as a
     subgoal to code the value and zero
     as arguments to the function
```

Figure 8: Examples of "Correct" and "Buggy" Rules Use by the LISP Tutor

tutorial in a rather mindless fashion by simply "guessing" at how to respond. The system, unlike a human tutor, has no way of detecting whether the student is just lazy or is genuinely confused.

The PUPs Architecture does not address these problems. It separates the solution generation and tutoring mechanisms of the LISP tutor, and generalizes them to programming languages. Although the LISP tutor and PUPS Architecture model "buggy" student behavior, they are not really concerned with providing an "exploratory" environment. Consequently they have few of the diagnostic problems of systems discussed earlier. One may wonder how much "deep" learning occurs with the LISP tutor. On the other hand, the LISP tutor is available in a non-experimental setting using current technology.

4.2.3 Coaching vs. Guiding

Coaching and guiding play two different roles in the tutoring process. Guiding seems most appropriate in introductory settings when there is a need to convey a particular method for doing something. Coaching seems more appropriate in settings where a number of different methods lead to the same solution. A good coach would highlight the distinctions between and advantages of various methods. Coaching and guiding also go hand in hand. A good coach occasionally has a complex point to make and might want to guide a student through a sequence of instructions. On the other hand, a good guide may occasionally want to do a bit of coaching, as Miller discovered while developing SPADE.

A robust tutoring system, and by extension, consulting system ought to be able to use both strategies. Students would be guided through a sequence of instructions that introduce new concepts or methods in a simple and straightforward manner. Later, as their expertise increases coaching strategies might be employed to highlight alternative or more sophisticated methods.

In developing such systems it will be important to keep in mind that in programming environments there is rarely one "best" way to do something. The choice of methods will depend both on the programming goal of the student and the pedagogical goal of the tutor. For example the buggy rule in Figure 8 is indeed a buggy method in the context of teaching about the function "ZEROP". But it is not always a buggy rule. The rule "works" in the LISP tutor because of the implicit pedegogical goal. In a broader context of general LISP programming it may not.

4.3 Tutoring Of Complex Skills Requires Analysis of Students' Plans

All of the tutoring systems discussed so far have attempted to build some sort of model of the student. This section will discuss the claim that tutoring complex procedural skills must take students' goals and plans into account. Systems like BUGGY, DEBUGGY and LMS are only able to locate misplaced or missing steps in a procedure. They are unable to diagnose *why* the wrong step was chosen. [Soloway *et al.* 83] found that similar diagnostic techniques were not enough for tutoring Pascal in open-ended environments. They suggested that good tutoring of programming seems to require a deeper analysis of students' problem solving processes. They developed a system called Proust that explored these issues [Johnson and Soloway 83]. Programming requires formulating plans. Therefore, tutoring programming ought to include analysis of students' planning strategies. [Genesereth 82] has made a similar argument for tutoring users of MACSYMA.

4.3.1 The MACSYMA Advisor Relies On Plans

[Genesereth 78] was the first to demonstrate the importance of plan analysis to tutoring complex procedural skills. He argued that the diagnostic techniques used by [Brown and Burton 78] that merely analyzed final answers were insufficient for more complex procedural skills. He was concerned with domains in which a correct answer can be produced by a number of different methods [Genesereth 82]. A procedural skill is viewed as a series of steps in a plan. If a student has a misconception about how to choose particular steps, the entire plan can go awry. Therefore, choosing a tutoring topic requires locating the misplaced step, which requires recognizing its inappropriate place in a plan.

Genesereth chose to study tutoring strategies in the domain of MACSYMA. MACSYMA is a large AI system that is used extensively by scientists, mathematicians and engineers to solve complex mathematical expressions. It is an interactive system based on an extensive command language. MACSYMA users begin by giving the system a mathematical expression. MACSYMA commands are sequentially chosen by the user to solve the expression. Each command is an instruction to use a standard mathematical technique to simplify the expression. In solving such expressions one cannot rely on rote methods. A solution is found through a judicious choice of commands. Unlike simple arithmetic where one learns explicit standard procedures, solving complex algebraic expressions requires *choosing appropriate operators.* As Genesereth argues, this can be viewed as *planning behavior.* MACSYMA was

an ideal domain in which to study this behavior because every "mental operation" is captured in the dialogue between the student and the system.

Genesereth points out that the MACSYMA commands are often misunderstood and consequently misapplied by new users. He observed that more expert users can usually locate such misunderstandings by noticing *faulty steps* in a novice user's plan. The MACSYMA Advisor was designed to mimic such *plan recognition* behavior. The major focus of the work was to construct suitable student models and explore plan recognition methods that could identify misconceptions.

The knowledge representation chosen was a dependency graph of both correct and incorrect knowledge about goals, plans, actions and effects. The system was able to trace students' behavior as a path through the dependency graph. Plan recognition required a goal and a set of actions as input. A plan was constructed by "starting at both ends". The actions constrain the choice of possible plans to those plans that include the actions. The goal and its subgoals restrict the potential effect of the actions. Misconceptions are located when an action produces an incorrect effect.

Genesereth demonstrated that a plan formalism was useful in locating misconceptions in complex problem solving environments. In particular, this work pertains to domains in which students' learning is focused primarily on the *primitive constructs* of a skill, rather than on the *procedures* that are built out of such constructs. In simple arithmetic one is more concerned with teaching complete procedures. For example most of us use a single simple rote method to do subtraction. On the other hand, solving algebraic expressions requires *choosing* a sequence of standard techniques.

The difference is in the emphasis on the procedure. In simple arithmetic it is more important to know a procedure than to know how to construct a procedure. In solving algebraic expressions it is more important to know how to construct procedures than to know them by rote. In simple arithmetic it may be sufficient to locate the bug and teach the correct step. In domains like MACSYMA it is important to explain *why* the step was incorrect, not simply that it was. The student must be able to use that particular step in the construction of other procedures.

4.3.2 Proust Finds Bad Plans in Pascal Programs

Constructing programs is a procedural activity. It is more like solving algebraic expressions than doing simple arithmetic. One begins with the basic constructs of a language and *builds* a program. Misconceptions about the basic constructs will affect how one combines the pieces of a program and will cause bugs [Johnson et. al. 83]. Proust [Johnson and Soloway 83] is an experimental system that identifies students' misconceptions about writing Pascal programs. Like the MACSYMA Advisor it uses knowledge of programming plans and goals to construct a

model of students' knowledge.

Studies of expert programmers revealed that they do not build programs from scratch, but rely on previously used plans [Soloway & Ehrlich 84]. [Bonar & Soloway 85] claim that novices create buggy programs through misapplication of planning strategies. Proust was designed to analyze Pascal programs and reconstruct the goals and plans used to create them. It has no actual tutoring capabilities.

Proust takes an abstract representation of a program specification and a Pascal program as input. It searches the program code for plans that satisfy the goals of the specification. Unlike MACSYMA, where steps in plans appear as sequential commands, plans in a program may overlap or be combined within the code. For example, in a program that calculates a variety of statistics on an array of numbers, a plan to keep a running total may be instantiated through an initialization at the top of the program, and an expression that increments a variable inside a loop which increments other variables as well. Therefore, reconstructing the plan of a Pascal programmer is significantly more complex than that of a MACSYMA user.

Proust relies extensively on domain knowledge about particular programming problems. Both correct and incorrect plans are encoded. The knowledge base contains an explicit *decomposition* of goals into subgoals and plans that satisfy them. Templates of Pascal code are attached to plans. Subgoals also contain information on how they relate to, and interact and interfere with other subgoals. The student's plan is constructed through a *prediction driven* evaluation process. A set of heuristics is used to choose the best potential subgoals. These restrict the way in which code fragments are associated with plans. A good potential plan predicts the purpose of the rest of the code. Proust is finished when all code fragments are accounted for as parts of good or bad plans.

Proust's diagnostic method is very successful in locating bugs in programs of real novice users. This is directly attributed to its use of goal/plan knowledge. That knowledge seems to be extremely domain dependent. PROUST can only analyze programs that implement a small set of specifications.

The research that contributed to the development of Proust should impact tutoring of procedural skills. Such skills are usually taught with an emphasis on isolated component parts. This is especially true of mathematics and programming. Even when students master the components, they find it difficult to integrate them. Proust highlights where integration strategies have failed. In order to correct such problems both initial teaching and remedial tutoring must put a greater emphasis on articulating explicit planning strategies.

4.4 Summary of Tutoring Systems

This section has introduced three important aspects of tutoring behavior. A tutor must be able to diagnose misconceptions. A tutor must also possess an explicit tutoring strategy such as coaching or guiding. A tutor, especially in a programming environment, must have knowledge about the possible goals and plans to accomplish them in the environment.

Understanding misconceptions is no small task. It requires representing expert knowledge. It also requires constructing a student model. All of the systems discussed relied to some extent on explicit knowledge of typical novice mistakes. Informal conversations with teachers indicates that this models human behavior. Human teachers do not diagnose student misconceptions from scratch, but rely on previous experience.

The tutoring strategy that is chosen seems to depend on the underlying philosophy of the system designers. Coaching is more appropriate in environments that encourage exploration. Guiding is more appropriate in more structured settings. Another perspective suggests that guiding is more appropriate for initial introductions. It is an efficient, organized way to present new material. Once new material has been introduced, the best way to learn it is to practice. This is especially true of procedural skills. Coaching is an ideal way to provide remediation. The student works independently as much as possible. The coach only intercedes when serious misconceptions are noticed.

Research on tutoring of complex procedural skills has led to greater insight into the kind of human problem solving that is used by programmers. Experts rely on previously solved problems. They do not solve problems from scratch. They adapt previously successful plans to new situations. The difference between an expert and a novice is the ability to choose good plans. The important distinction to make is between skills that require following rote procedures and those that require integrating constructs in innovative ways. In environments where the latter occurs, tutors must possess comprehensive knowledge of goals and plans. The next section includes a disccusion of how tutoring skills can be encorporated into a programming enviroment.

5 A Summary of Consulting Behavior

In the preceding sections automated consulting in programming environments has been characterized in three ways. Relief from mundane detail is often available directly within an environment. Mechanisms for obtaining reference information about commands and constructs are usually provided by on-line help facilities. Although only in the earliest stages of development, tutoring systems diagnose user misconceptions through tutoring strategies that coach or guide. Two themes have emerged, namely that most consulting behavior is geared toward either novices or experts but not both, and that good automated consulting must take users' computational goals and plans into

account. This section will review these themes, articulate inherent problems, and suggest some solutions.

Table 2 summarizes the systems that were discussed in this paper. With the exception of BUGGY and LMS, they are concerned with using a computing environment to do something or teach something. All but WUSOR, WEST and the MACSYMA Advisor focus on the task of programming. The table lists the intended user of each system, the philosophical perspective of the environment and the ability of the system to assist with planning. Users are characterized as expert or novice. Environments are characterized as exploratory or structured. Exploratory environments encourage users to develop solutions to problems in their own way. Structured environments impose a design discipline that provides needed guidance, especially in product oriented settings. Some systems can directly assist users in reasoning about their plans and goals. Others provide indirect plan level assistance through plan-based structured environments.

From table 2 one can see the relationships between the type of consulting behavior, the intended users, the environment and the ability to assist with planning. Managerial assistance is primarily for experts, tutoring is for novices, and on-line help is for either. All three types of consulting behavior occur in either exploratory or structured environments. All three types also include systems that attempt to assist with planning.

A more careful examination of the table highlights the following important issues:

- Managerial assistance and the programmer's apprentice are primarily for experts, as might be expected. Both require that the user has "shared knowledge" [Waters 85] with the system, and can evaluate what the system does behind the scenes.
- Non-textual interaction relieves the user of syntactic detail through techniques that reflect the distinction between structured and exploratory programming. Structure editors monitor and restrict the use of data abstractions and processes, and encourage a systematic programming style. Viewing systems provide a single concrete conceptualization that can be adapted to the task at hand within an exploratory environment.
- The type of information provided by on-line help, and the means for accessing it is also dependent upon the expertise of the user. Command languages and definitions are primarily for experts since they will be most familiar with the language and conventions of a system. Canned explanations and menus assume less about users' knowledge of such conventions and consequently aid novices in their search for information. Natural language based systems that analyze user plans go a step further by doing the search for information for users.
- The nature of the environment relates to assistance with planning. Structured environments implicitly guide planning by imposing a design discipline. The systems that explicitly help with planning do so in exploratory environments in which individual styles are encouraged. The important point is that assistance at a purely syntactic or command level is insufficient. Consulting must occur at a goal/plan level too.

Two fundamental problems have emerged:

1. Novices and experts have different consulting needs, but it is very difficult to categorize users as one or the other. Furthermore, since they have different needs, how can a system provide mechanisms that

Type of Help	*System*	*Intended Users*	*Environment*	*Helps w/ Planning*
Relief From Mundane Detail				
Managerial Assistance	Interlisp	Experts	Exploratory	No
	Gandalf	Mostly Experts	Structured	Indirectly
Non-textual Interaction				
Structure Editors	Cornell P. S.[16]	Novice	Structured	Indirectly
	EMACS	Expert[17]	Exploratory	No
Viewing Systems	Smalltalk	Both	Expl. w/ Model	No
	Boxer	Novice	Expl. w/ Model	No
Programmer's Apprentice	KBEmacs	Expert	Exploratory	Yes
Information Provided Through On-Line Help				
Canned Definitions	Commercial			
Commands and Keys	Systems[18]	Expert	Both	No
Menus	"	Either	Both	No
Canned Explanations				
Commands & Keys"	Either	Both	No	
Menus	"	Novice	Both	No
Natural Language	Consul/Cue	Both	Exploratory	No
	WIZARD	Novice	Exploratory	Yes
	UC	Novice	Exploratory	Yes
Techniques for Tutoring				
Surface Misconceptions	BUGGY	Novice	---	No
	LMS	Novice	---	No
Tutoring Strategies				
Coaches	WUSOR	Novice	Exploratory	No
	WEST	Novice	Exploratory	---
Guides	Spade	Novice	Structured	Indirectly
	LISP Tutor	Novice	Structured	Indirectly
Underlying Misconceptions	Macsyma Adv.	Novice	Exploratory	Yes
	Proust	Novice	Exploratory	Yes

--- indicates not applicable

Table 2: Summary of Systems Surveyed

allow a novice to become an expert?

2. Environments that provide structure inhibit individual style, but exploratory ones offer little guidance. How can the positive aspects of both be exploited within a single environment?

The novice/expert categorization simplifies the design process, but doesn't really address user needs. Users rarely fall cleanly within the ranks of novice or expert. Depending on the tasks they do, and the facilities they learn

about, users have varying expertise with components of the system. Users with extensive experience may still require an introduction to a facility they have never used. In particular instances beginners can often benefit from a sophisticated feature, provided they can understand it.

Both kinds of environments can be useful to both novices and experts. Structured environments provide an efficient means for novices to learn and be reminded about the commands and constructs of an environment. Experts benefit from structure when the primary objective is to efficiently produce robust systems. Exploratory environments on the other hand can let novices develop their own style of problem solving. The inherent flexibility and extensibility of such environments is appropriate for experts when experimenting with and articulating a problem is more important than its solution. Therefore, structured environments seem more appropriate when efficiency is a priority, while exploratory environments are more appropriate when creative freedom is more important.

In order to address the problems described above, we a layered, user goal centered approach to the design of automated consulting facilities in programming environments is needed. Components of this approach have been articulated by others [diSessa 85; Paris 86; Adelson & Soloway 84]. The core of such an environment should be exploratory in nature, with judiciously added layers of structure. The design should rely on a principled underlying conceptual model [diSessa 85] to which all aspects of the system adhere. The basic data and process abstractions within the environment should be defined only after a careful articulation of the goals that can be accomplished within the environment and the plans one uses to accomplish them [Adelson & Soloway 84]. Finally, instead of designing interactive mechanisms that distinguish novices from experts a single cohesive approach should focus on what users want to do with and know about the environment [Paris 86].

An environment that is fundamentally exploratory in nature would consist of a set of tools that can be manipulated in complex ways and extended and customized by users who are exploring new problems. In order to gain entry into such an environment, carefully designed mechanisms would structure and guide less experienced users. Such mechanisms could also assist users who want to rely on a particular design discipline. These mechanisms would provide default methods for accomplishing standard computational goals using simple, straightforward plans. Users who are satisfied with the default methods would never need concern themselves with more sophisticated and flexible ones. User who require more freedom would peal back layers of structure to expose the underlying tools.

A user goal centered approach would provide degrees of freedom that depend on tasks, rather than on the system as a whole. In this way, a user need not graduate to an "experts" system. Some tasks are accomplished by

relying on imposed structure, others are done in a more open environment. Similarly, the mechanisms required for consulting purposes should also be organized in terms of goals and plans, and should be viewed as an integral part of the environment. Consulting behavior should be targeted for specific goals, rather than toward individual commands. Users are only required to learn about the features of the system that pertain to their immediate goal. Specifically, consulting resources should:

- Introduce new aspects of the system by including tutorials that guide.
- Clear up user misunderstandings by including tutorials that coach.
- Clarify details and options about features that the user has not yet learned, by including tutorials that guide or coach depending on the degree to which the information is new.
- Remind the user of how to do things by including easily accessible definitions and/or examples.
- Do things for the user when the system and the user agree fully on the nature of the task.

We do not claim that developing such an environment and its consulting facilities is an easy task. The main focus of such development must concentrate on how the environment can be used effectively. In other words it must take users' computational goals into account. Furthermore, it must balance extensibility and self expression with truly useful support structures that are accessible to beginners as well as more experienced users. Most of the design considerations discussed in this section have been successfully incorporated into the systems that were described earlier in this paper. The next step in the design of environments will require integrating the best of these ideas, and carefully considering how consulting mechanisms can meet the needs of a variety of users.

References

[Abelson & diSessa 81]
Abelson, H and A. diSessa.
Turtle Geometry: The computer as a medium for exploring mathematics.
MIT Press, Cambridge, MA, 1981.

[Adelson & Soloway 84]
Adelson, B and E. Soloway.
A Model of Software Design.
Technical Report CSDRR-342, Yale University, New Haven, Connecticut, 1984.

[Anderson 86] Anderson, J.R. and E. Skwarecki.
The automated tutoring of introductory computer programming.
Communications of the ACM 29(9):842-849, September, 1986.

[Apple Writer IIe 82]
Apple Writer IIe.
Apple Writer IIe Reference Manual.
Apple Computer Inc., Cupertino, California, 1982.

[Barstow 84] Barstow, D.R.
A display oriented device for INTERLISP.
Interactive Programming Environments.
McGraw Hill, New York, 1984.

[Bonar & Soloway 85]
Bonar, J. and E. Soloway.
Preprogramming knowledge:A major source of misconceptions in novice programmers.
Human-Computer Interaction 1(2):133-161, 1985.

[Borenstein 85] Borenstein, N.S.
The design and evaluation of on-line help systems.
PhD thesis, Carnegie Mellon University, April, 1985.

[Brooks 75] Brooks, F.P.
The Mythical Man-Month: Essays on Software Engineering.
Addison-Wesley Publishing Co., Reading, MA, 1975.

[Brown & VanLehn 80]
Brown, J.S. and K. VanLehn.
Repair theory: A Generative theory of bugs in procedural skills.
Cognitive Science 4:379-415, 1980.

[Brown *et al.* 82] Brown, J.S., R.R. Burton and J. deKleer.
Pedagogical, natural language and knowledge engineering techniques in SOPHIE I, II and III.
Intelligent Tutoring Systems.
Academic Press, London, 1982, pages 227-281.

[Brown and Burton 78]
Brown, J.S. and R.R. Burton.
Diagnostic Models for Procedural Bugs in Mathematics.
Cognitive Science 2:155 - 192, 1978.

[Browser 83] BROWSER.
NMODE Reference Manual.
Hewlett-Packard Company, Cupertino, California, 1983.

[Burton 82] Burton, R.R.
Diagnosing bugs in a simple procedural skill.
Intelligent Tutoring Systems.
Academic Press, London, 1982, pages 157-182.

[Burton & Brown 82]
Burton, R.R. and J.S. Brown.
An investigation of computer coaching.
Intelligent Tutoring Systems.
Academic Press, London, 1982, pages 79-98.

[Chin 86] Chin, D.N.
User modeling in UC, the UNIX Consultant.
In *Proceedings of the CHI'86 Conference*, pages 13-17 . Boston, MA, April, 1986.

[Clancey 82] Clancey, W.J.
Tutoring rules for guiding a case method dialogue.
Intelligent Tutoring Systems.
Academic Press, London, 1982, pages 201-225.

[Dahl *et al.* 76] Dahl, O.J., E. Dijkstra & C.A.R. Hoare.
Structured Programming.
Academic Press, London, 1976.

[Dijkstra 76] Dijkstra, E.
A Discipline of PRogramming.
Prentice-Hall, Englewood Cliffs, NJ, 1976.

[diSessa 85] diSessa, A. A.
A principled design for an integrated computational environment.
Human-Computer Interaction 1:1-47, 1985.

[diSessa & Abelson 86]
diSessa, A. A. and H. Abelson.
Boxer: A reconstructible computational medium.
Communications of the ACM 29(9):859-868, September, 1986.

[Dolotta *et al.* 84] Dolotta, T. A., R. C. Haight and J. R. Mashey.
UNIX Time-Sharing System: The Programmer's Workbench.
Interactive Programming Environments .
McGraw Hill, New York, 1984, pages 253-2369.
first published in 1972.

[Donzeau-Gouge *et al.* 84]
Donzeu-Gouge V.,G Huet, G. Kahn & B. Lang.
Programming Environments Based on Structured Editors: The MENTOR Experience.
Interactive Programming Environments .
McGraw Hill, New York, 1984, pages 128-140.

[Ellison & Staudt 85]
Ellison, R.J. & B.J. Staudt.
The Evolution of the GANDALF System.
The Journal of Systems and Software 5(2):107-120, May, 1985.

[Finin 83] Finin, T.
Providing Help and Advice in Task Oriented System.
In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, pages 176 - 178. International Joint Conference on Artificial Intelligence, Karlsruhe, West Germany, 1983.

[Genesereth 78] Genesereth, M.R.
Automated Consultation of Complex Computer Systems.
PhD thesis, Harvard University, 1978.

[Genesereth 82] Genesereth, M.R.
The role of plans.
Intelligent Tutoring Systems.
Academic Press, London, 1982, pages 137-155.

[Goldberg 84] Goldberg A.
The Influence of an Object-Oriented Language on the Programming Environment.
Interactive Programming Environments.
McGraw Hill, New York, 1984, pages 141-174.

[Goldberg & Robson 83]
Goldberg, A. and D. Robson.
Smalltalk-80, The language and its implementation.
Addison Wesley, Reading, MA, 1983.

[Goldstein 82] Goldstein, I.R.
A genetic graph model for tutoring.
Intelligent Tutoring Systems.
Academic Press, London, 1982, pages .

[Groen 79] Groen, G.J.
The theoretical ideas of Piaget and educational practice.
Impact of Research on Education: Some Case Studies.
National Academy of Education, Washington D.C., 1979.

[Habermann & Notkin 86]
Habermann, A.N. and D. Notkin.
Gandalf: Software development environments.
IEEE Transactions on Software Engineering SE-12(12):1117-1127, December, 1986.

[Johnson and Soloway 83]
Johnson W. L. and E. Soloway.
PROUST: Knowledge-Based Program Understanding.
Technical Report YaleU/CSD/RR#285, Yale University, Department of Computer Science, 1983.

[Johnson et. al. 83]
Johnson W. L., E. Soloway, B. Culter and S. Draper.
Bug Catalogue: I.
Technical Report YaleU/CSD/RR#285, Yale University, Department of Computer Science, 1983.

[Kaczmarek & Sondheimer 83]
Kaczmarek, T., W. Mark, and N. Sondheimer.
The Consul/CUE Interface: An Integrated Interactive Environment.
In *CHI'83 Proceedings*, pages 98-102. , 1983.

[Kaiser & Kant 85]
Kaiser G. E. and E. Kant.
Incremental Parsing without a Parser.
The Journal of Systems and Software 5(2):121-144, May, 1985.

[Kaplan 82] Kaplan, S.J. .
Cooperative responses from a portable natural language database query system.
Artifical Intelligence 2(19), 1982.

[Kay & Black 85] Kay, D.S. and J.B. Black .
The evolution of knowledge representations with increasing expertise in using systems.
In *Proceedings of the Seventh Annual Conference of the Cognitive Science Society*, pages 140-149. Cognitive Science Society, Irvine, CA, August, 1985.

[Levinson 83] Levinson, S.
Pragmatics.
Cambridge University Press, Cambridge, England , 1983.

[Magers 83] Magers, C. S.
An experiemntal evaluation of On-line HELP for non-programmers.
In *CHI'83 Proceedings*, pages 277-281. 1983.

[Matthews 84] Matthews, K.
Taking the initiative in problem-sovling discourse.
Technical Report CUCS-114-84, Columbia University , New York, NY, 1984.

[Matz 82] Matz, M.
Towards a process model of high school algebra errors.
Intelligent Tutoring Systems.
Academic Press, London, 1982, pages 25-49.

[Mays 80] Mays, E.
Correction misconceptions about data base structure.
In *Proceedings 3-CSCSI.* Canadian Society of Computational Studies of Intelligence, May, 1980.

[McCoy 83] McCoy K.F.
Correcting Misconceptions: What to say when the user is mistaken.
In *Proceedings of the CHI'83.* 1983.

[McKeown *et al.* 85]
McKeown, K.R., Wish, M. and Matthews, K.
Tailoring explanations for the user.
In *Proceeding of the IJCAI.* 1985.

[Miller 82] Miller, M.L.
A structured planning and debugging environment for elementary programming.
Intelligent Tutoring Systems.
Academic Press, London, 1982, pages 119-135.

[Papert 72] Papert, S.
Teaching children to be mathematicians vs. teaching about mathematics.
International Journal Math. Educ. Sci. Technology 3:249-262, 1972.

[Papert 80] Papert, S.
Mindstorms: Children, computers and powerful ideas.
Basic Books, 1980.

[Paris 85a] Paris, C.
Description strategies for naive and expert users.
In *Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics.*
Chicago, IL, 1985.

[Paris 85b] Paris, C.
Towards more graceful interaction: A survey of question answering programs.
Technical Report, Department of Computer Science, Columbia University, New York, NY, 1985.

[Paris 86] Paris, C. L.
Tailoring Object Descriptions to the User's Level of Expertise.
Paper presented at the International Workshop on User Modelling, Maria Laach, West Germany.
August, 1986

[Pollack 86] Pollack, M.
Inferring domain plans in question-answering.
PhD thesis, Moore School, University of Pennsylvania, May, 1986.

[Rich & Shrobe 78]
Rich C. and H. E. Shrobe.
Initial Report on a LISP Programmer's Apprentice.
IEEE Transactions of Software Engineering SE-4(6):456-467, November, 1978.
reprinted in *Interactive Programming Environments.*

[Rissland *et al.* 80]
Rissland, E.L., Valcarce E. M., and C. R. Perrault.
Explaining And Arguing With Examples.
???? 0(0):288-294, 1980.
Article from Kathy's class, no journal title!

[Sandewall 78] Erik Sandewall.
Programming in an Interactive Environment:The LISP Experience.
Communications of the ACM 10(1):35-71, March, 1978.
reprinted in *Interactive Programming Environments.*

[Sheil 84] B. A. Sheil.
Power Tools for Programmers.
Interactive Programming Environments.
McGraw Hill, New York, 1984, pages 19-30.

[Shrager & Finin 82]
Shrager, J. and T. Finin.
An expert system that volunteers advice.
In *AAAI82*, pages 339-340 . 1982.

[Sleeman 82] Sleeman, D.
Assessing aspects of competence.
Intelligent Tutoring Systems.
Academic Press, London, 1982, pages 185-199.

[Sleeman & Brown 82]
Sleeman, D. and J.S. Brown (editor).
Intelligent Tutoring Systems.
Academic Press, London, 1982.

[Smith *et al.* 85] Smith, D. R., G. B. Kotik and S. J. Westfold.
Research on Knowledge-Based Software Environmnets at Kestrel Institute.
IEEE Transactions on Software Engineering SE-11(11):1278-1295, November, 1985.

[Solomon 86] Solomon.
Computer environments for children, a reflection on theories of learning and education.
MIT Press, Cambridge, MA, 1986.

[Soloway & Ehrlich 84]
Soloway E., and K. Ehrlich.
Empirical studies of programming knowledge.
IEEE Trans. Softw. Eng. 5(SE-10):595-609, 1984.

[Soloway *et al.* 83]
Soloway, E., E. Rubin, B. Woolf, J. Bonar and W. L. Johnson.
MENO-II: An AI-Based Programming Tutor.
Journal of Computer-Based Instruction 1(1):20-34, 1983.

[Stallman 81] Stallman, R.M.
Emacs The extensible, customizable, self-documenting display editor.
In *SIGPLAN SIGOA Symposium on Text Manipulation*, pages 147-156. June, 1981.

[Stevens *et al.* 82] Stevens, A., A. Collings and S. E. Goldin.
Misconceptions in students' understanding.
Intelligent Tutoring Systems.
Academic Press, London, 1982, pages 13-24.

[Teitelbaum & Reps 81]
Teitelbaum T. and T. Reps.
The Cornell Program Synthesizer: A Syntax-Direct Programming Environment.
Communications of the ACM 24(9):563-573, September, 1981.
reprinted in *Iteractive Programming Environments.*

[Teitelman 84a] Teitelman, W.
Automated Programmering: The Programmer's Assitant.
Interactive Programming Environments.
McGraw Hill, New York, 1984, pages 232-239.
first published in 1972.

[Teitelman 84b] Teitelman, W.
A Display-Oriented Programmer's Assistant.
Interactive Programming Environments.
McGraw Hill, New York, 1984, pages 240-287.
first published in 1972.

[Teitelman & Masinter 81]
Teitelman W. and L. Masinter.
The Interlisp programming environment.
Computer 14(4):25-34, April, 1981.

[Tops-20 84] Tops-20 Help.
HELP.HLP.
On-line help, CUCS20, Columbia University, NY, NY, 1984.

[VanLehn 83] VanLehn, K.
Human procedural skill acquisition:Theory, model and psychological validation.
In *AAAI'82*, pages 420-423. 1983.

[Waters 82] Waters, R. C.
The PRogrammer's Apprentice: Knowledge Based Program Editing.
IEEE Transactions on Software Engineering SE-8(1):??? , January, 1982.
reprinted in *Interactive Programming Environments*.

[Waters 85] Waters, R.C.
The programmer's apprentice: A session with KBEmacs.
IEEE Transactions on Software Engineering 11:1296-1320, 1985.

[Waters 86] Waters, R.C.
KBEmacs: Where's the AI?
The AI Magazine 7(1):47-56, Spring, 1986.

[Wilensky 83] Wilensky, R.
Planning and understanding.
Addison-Wesley, Reading, MA, 1983.

[Wilensky *et al.* 84]
Wilensky, R., Y. Arens, and D. Chin.
Talking to Unix in english:An overview of UC.
Communications of the ACM 27(6):574-593, June, 1984.

[Word-Star 80] .Word-Star.
Word-Star User Reference Manual.
MicroProp International Corp., San Rafael, California, 1980.

[Xinfo 84] XINFO.
On-line Help, CUCS20.
On-line help, CUCS20, Columbica University, NY, NY, 1984.