Rapid Prototyping

H. Rex Hartson
Eric C. Smith

TR 87-26

OUTLINE

INTRODUCTION

The Concept of Prototyping.

In a television interview (CBS 1986) Anthony Perkins described a technique used by Alfred Hitchcock for developing and refining the plots of his movies. Hitchcock would tell the stories at cocktail parties and observe reactions of his listeners. He would experiment with various sequences and mechanisms for revealing the story line. Refinement of the story was based on listener reactions' as an evaluation criterion. *Psycho* is one notable example of the results of this technique.

Auto makers, architects, and sculptors make models; circuit designers build "bread-boards"; aircraft developers test prototypes; artists experiment with working sketches. In each case the goal is to provide an early ability to observe something about the nature of the final product, evaluating ideas and weighing alternatives before committing to one of them.

In contrast, conventional approaches to development of large interactive software systems — a highly complex process that requires enormous quantities of time, money, and personnel — forces a commitment to large amounts of design detail without any means for visualizing the result until it is too late to make significant changes. It is little wonder that there is so much end-user dissatisfaction with many of the products so developed.

Recently, however, the techniques of prototyping and iterative refinement have emerged in the context of software development, especially for the human-computer interface. The subject of this chapter is a technique for interface and software development referred to as *rapid prototyping.* It, too, has the goal of early observation of behavior. In addition it shares the essence of the Hitchcock story development scheme, refinement of the product based on feedback from users.

One last point should be made concerning the Hitchcock approach: in spite of the vast difference between prototype and finished product (i.e., verbal storytelling versus motion picture), the prototyping technique was used to great effect by a master dialogue designer.

The *prototyping* approach to interactive software system design involves the production of at least one early version of the system that illustrates the essential features of the later, operational system. With rapid prototyping, the process of constructing system prototypes is accelerated, so that the time from beginning a prototype to evaluating end-user responses is much shorter. This, in turn, allows multiple iterations through the refinement process and a finer tuning to the needs of the end-user, leading to a high degree of confidence in the usability of the resulting system.

The technique of rapid prototyping may be applied to the design of any part of a system. However, the focus here is on user-oriented issues such as learnability, usability, and functionality; rapid prototyping of human-computer interfaces will be emphasized in this chapter. Often this can mean prototyping only the interface portion of a system. Computation of results, storage and retrieval of information, and other tasks not directly observable by the end-user can be stubbed into the prototype, saving time in its construction.

Weighing Rapid Prototyping.

Advantages. Use of prototypes in the design and implementation of software

systems represents a significant departure from traditional development techniques. To justify such a change in practice, some substantial benefits must be obtainable. There are a few experimental studies on the subject of prototyping *versus* classical system development methods in which benefits to both developers and end-users are cited.

In an experiment conducted at UCLA (Boehm, Gray, and Seewaldt 1984), some development teams used conventional development methodologies while others employed prototypes in the development process. Systems produced by the groups using prototypes were judged to be easier to learn and use than those produced by standard methods. Groups using the prototyping approach also appeared to be less affected by deadline pressures. Code of the final systems produced by prototyping groups was only about 40 percent as large as that of their counterparts. Finally, the prototyping groups accomplished their task with 45 percent less effort than the other groups.

In a similar study (Alavi, 1984), responses from end-users of the systems produced in the experiment indicated that end-users of systems developed using the prototyping approach were better disposed toward the product than were users of non-prototyped systems. Software developers who had been involved in product development using prototypes felt that use of prototypes provided a valuable means for understanding

what end-users really want in a proposed system. One developer commented that "The end-users are extremely capable of criticizing an existing system but not too good at articulating or anticipating their needs." Developers also felt that prototyping enhanced communication about the proposed system. The prototype created a common baseline or reference point from which potential problems and opportunities could be identified. Discussions could take place between designers and end-users about good and bad features in the evolving design. The prototype allowed these discussions to be conducted in concrete terms.

Users also tended to be more enthusiastic about a project in which they were involved through the use and evaluation of prototypes. According to the developers, this enthusiasm, together with the enhanced communication of requirements, led to increased end-user acceptance of the systems. The first version that end-users can experiment with, whether prototype or end product, can cause them to change their view about what they want the system to do (Wasserman and Shewmake 1982). Use of prototypes in design evaluation can respond to these changes earlier and can increase the likelihood that the end product will be what users really want.

The advantage that *rapid* prototyping has to offer in addition to the prototyping concepts in the above studies is that of iterative refinement. In each of the studies

mentioned above, prototypes were manually coded by the designers. Due to time constraints there was little opportunity for multiple passes through the prototype phase of development. The rapid prototyping technique can be enhanced by automated tools that allow developers quickly to record the design of important components of the proposed system — documenting its behavior, especially that of the interface. As much as possible non-coding techniques, such as direct manipulation construction of interface displays and state diagram representations of logical sequencing, are used to represent the design. This representation of the design can then be executed and used as a prototype. A prototype, however embryonic, can be available for experimentation and evaluation very early in the development cycle. Changes in the prototype can be made rapidly using the same design representation tools. Because the tools allow rapid representation of design ideas, end-users can be presented with many options instead of a single design, increasing their ability to maneuver toward a design which meets their needs. Because of the many advantages of rapid prototyping, it is difficult to avoid the conclusion that no interactive system ought to be produced without at least a simple paper and pencil prototype, evaluated with user feedback.

Pitfalls. Prototyping, however, is not without potential drawbacks as an approach to interactive system development. These are mostly pitfalls, rather than disadvantages; with some caution they can be avoided.

One of the biggest dangers is found in attempts to use prototyping as a development technique without first securing cooperation from the parties involved and without establishing a thorough understanding of the process. First, iterative refinement depends on the willingness and ability of customers and end-users to provide useful feedback. Also, established management procedures can make it difficult to deal with planning and scheduling of a development life cycle quite different from the traditional one. Managers may view as wasteful the application of resources to building a prototype. System developers themselves also must have the proper attitude. For example, Alavi (1984) noticed a reduction in programmer discipline, possibly because the process was viewed as an exercise rather than as "the real thing". Also, prototypes of large systems can be large in themselves. The misconception that a prototype is just a toy can lead to its development without a methodology to aid in its management, resulting in a failed, unmanageable project. These problems can be addressed by methodologies and tools built around a prototyping-based approach (see the section on "System Development Issues").

Some of the most serious problems can occur if various parties begin to view the prototype as the final system. End-user and developer enthusiasm for continued development may diminish after a "working" prototype is provided (Alavi, 1984). Managers, upon seeing the prototype, can be tempted to rush it prematurely to the market —

often to the astonishment and frustration of the developers. Both cases are abuses of the prototyping technique and represent management problems that can be avoided by having an early agreement about the role of prototyping in the overall development process.

Prototypes with emphasis on the end-user interface usually have a bottom-up flavor to their development, because details of the interface design tend to surface early. It can be difficult for a software engineer trained in the ways of top-down, step-wise decomposition to accept such a different approach to the interface portion of the system. Also, emphasis on the interface in the prototype can lead to stubbing of computational functionality. The temptation is to stub the difficult parts of the computational design without first understanding their design requirements. Later, development of the stubbed functions can reveal basic problems that affect the system at many levels above the stub in question. The effect can even reach the interface component. The result is upheaval rather than a smooth progression toward an implementation. This kind of problem is a good reason for mixing some bottom-up development with the top-down step-wise decomposition process of the computational software of the design. Another reason is the development of error handling (McFarland, 1986). Strict adherence to a top-down approach makes it difficult to specify an accurate description of a system's error handling functions.

A Natural Technique.

Although prototyping, especially rapid prototyping, has been closely associated in the literature with automated tools, it is important to recognize that *prototyping is a technique, not a tool.* The technique can be effective even when performed manually (e.g., as part of a paper and pencil exercise), especially in the early, conceptual stages of development. Furthermore, since it is a technique that begins with specific details of an interface design, then structures and refines them into a system, there are sound theoretical reasons for believing that this is a *natural technique*, grounded in the precepts of developmental psychology (Piaget, 1952; Whiteside and Wixon, 1985). Working from concrete to abstract is the way humans naturally investigate new concepts and solve problems. To both end-users and developers, a prototype is concrete while specifications are abstract. Rapid prototyping is also part of the notion of iterative refinement. It is not that designers must be afforded a chance to be lazy or sloppy with the initial design, but it is simply not possible, using design principles alone, to get it right the first time. They are thus forced to adopt the "artillery method": Ready, fire, aim! The first shot serves to provide a reference point from which adjustments are made in order to hit the target. As applied to the development of interactive systems, the rapid prototyping approach is changing the way these systems are developed. There

is great potential to make the development process faster and the product better and more usable.

Not only is rapid prototyping a natural technique, but it is highly suitable for the special situation in which various parties of the development team find themselves. Development of interactive systems must be a cooperative effort between behavioral scientists and computer scientists (Hartson, 1985). A gap exists between the skills and goals brought to the task by each of these roles. Computer scientists often do not fully understand the need for user-centered design and human factors, and how they are achieved. Alternatively, human factors engineers often do not appreciate the constraints and difficulties of building large interactive systems and of integrating the user interface with the rest of the software.

The behavioral scientist, trained in analysis and evaluation, is now part of an environment primarily intended for synthesis and design. That environment must, however, include more analysis and evaluation. This is not just a temporary situation, either, until we learn how to do it right the first time. Because, as Carroll and Rosson (1985) state, design activity is essentially empirical "...not because we don't know enough yet, but because in a design domain we can never know enough". System design is inherently more art than science, and art is where analysis meets synthesis

because the possibilities are infinite. The two developer roles must work together to achieve an artful result.

The primary function of human factors work is testing. But at the beginning of the design cycle there is nothing to test, a dilemma for the behavioral scientist. Building a system to test is expensive and time consuming and is a large investment in design concepts that have not been evaluated; thus the dilemma affects the computer scientist, too. The needs and constraints of each role work conflict with those of the other role. Through rapid prototyping, an early opportunity is afforded the human factors engineer to observe and evaluate system behavior. By building ease of modification into the prototype the computer scientist is providing *human factorability*. Rapid prototyping is an important factor in harnessing the sometimes opposing forces of these roles and helping them work together.

Related Work.

Construction and modification of software by ordinary programming techniques are notoriously expensive and time consuming activities. Since prototyping involves

construction and modification of a software model of a system, it should not be surprising that much rapid prototyping work to date has been involved with the construction of special prototype definition and execution environments. These environments attempt to allow designers to construct useful prototypes while reducing the amount of conventional programming required. In the late 1970's and early 1980's several such environments and tools were developed. These have served as examples and starting points for much of the current research in rapid prototyping.

ACT/1. ACT/1 (Mason and Carey, 1981; Mason and Carey, 1983), developed by Art Benjamin and Associates of Toronto, is one of the first commercially available products for rapidly prototyping end-user interface scenarios. ACT/1 employs a specification-by-example technique that allows designers to create interface screens by filling in parts of a screen. Procedural links are specified in tabular form with entries having the format:

$$< \text{input screen, process, output screen} >.$$

At first, with no application logic specified, end-users may go through a fixed script simulation of the end-user interface. Application logic can be added to create a first prototype of a new system. ACT/1 has had more than one hundred end-users and has been applied to the development of several interactive information systems.

FLAIR. The Functional Language Articulated Interactive Resources (FLAIR) system was developed at TRW to aid designers in involving users in the design process (Wong and Reid, 1982). FLAIR facilitates design of interfaces based on hierarchies of menus. It allows simulation and experimentation with such hierarchies. FLAIR's prototyping abilities are largely restricted to the interface portion of the system, producing elaborate graphical facades. FLAIR was among the first to provide a Dialogue Design Language (DDL). The DDL, with its voice-driven menu interface, is used to describe the end-user interface, rather than formal grammars that are often used.

IDS. The Interactive Dialogue Synthesizer (IDS) was developed at Martin Marietta as a tool to aid in the production of interfaces for command and control systems (Hanau and Lenorovitz, Proceedings, 1980; Hanau and Lenorovitz, SIGGRAPH, 1980). The IDS uses Backus-Naur form rules to define the interaction language for the target system. Displays are attached to the grammar as semantic actions. These displays, which represent "snapshots" of the final system, can then be used by a simulator to give the end-user a feel for how the target system will eventually behave. IDS is a good example of a tool designed specifically to support rapid prototyping. Information gathered through the use of simulation may be quickly integrated into a new version of the prototype, because of the very high level of interface definition. No programming

is necessary to alter the form, appearance, or position in sequence of a part of the interface.

RAPID/USE. RApid Prototypes of Interactive Dialogues (RAPID) is a tool designed to be used with the User Software Engineering (USE) methodology (Wasserman and Shewmake 1985; Wasserman, Pircher, Shewmake, and Kersten 1986) in the context of interactive information systems. RAPID/USE relies on state transition diagrams for defining interaction languages. Displays are associated with state nodes and input with state transition arcs. Dialogue, the contents of the nodes, is programmed with a high level textual dialogue definition language. Prototype interfaces can be defined and simulated rapidly. As the interface prototype becomes more stable, prototype application semantics may be attached using the Troll/USE database manipulation package. Eventually a fully operational prototype can be created. The USE methodology is one of the first to provide explicitly for use of prototypes in the design phase. Design/prototype iteration is specifically included in the life cycle.

Behavioral Demonstrator. The Behavioral Demonstrator (BD) (Callan, 1985) is intended to support rapid prototyping within the Virginia Tech Dialogue Management System. The Behavioral Demonstrator interprets designs represented as supervised flow diagrams, which describe high level flow of control in a target system. Dialogue

content is created using specialized direct manipulation tools. Computational functionality is either programmed or stubbed in. A support environment is provided for executing partially specified and incompletely developed designs. As the design matures and becomes complete, the prototype evolves into a real, compilable implementation of the entire target system. The goal of the Behavioral Demonstrator is to provide systems and interface developers with the ability to alter the design during the running of the prototype, and to restart from that same point in its execution, providing very rapid turn-around from concept to example.

The Rapid Intelligent Prototyping Laboratory. The Rapid Intelligent Prototyping Laboratory (RIPL), developed at Computer Technology Associates in Englewood, Colorado, is a set of hardware and software tools to support construction of facade prototypes for complex interactive systems (Flanagan, Lenorovitz, Stanke, and Stocker 1985). Interface components called "tiles" are created by the designer using a set of direct manipulation tools. A "Simulation Subsystem" links these tiles and user-defined routines together to simulate the system. RIPL employs two expert systems to aid in interface construction. A "consultation expert" provides design time advice to interface designers, and an "Evaluation Expert" is used to evaluate the prototype itself.

KINDS OF PROTOTYPES

There are at least three (more or less orthogonal) dimensions along which various approaches to prototyping can be classified. These three dimensions, shown in Figure 1, are:

- revolutionary versus evolutionary,
- interface only versus whole system, and
- intermittent versus continuous.

In the first dimension, a distinction is made regarding how the prototype stands in relation to the final product. In one direction of this dimension there is a *revolutionary* development process in which a prototype is designed, built, evaluated, and scrapped before work begins anew on the real system. That is, the prototype is disposable. In the other direction there is an *evolutionary* development process in which a prototype,

through iterative modification, evolves into a complete implementation of the final target system.

The second dimension separates those prototypes that address the end-user *interface only* from those that represent the *whole system*, including its computational component. Interface only prototypes are very common; a mock-up facade is fairly easy to construct and execute. Conversely, whole system prototypes are difficult to build; their support environment requires much more technically complicated support.

In the third dimension distinction is made between intermittently executable and continuously executable prototypes. Prototypes for which the ability to demonstrate system behavior is only *intermittent* can be exercised only at times in the development process when a particular version of the system has been completely constructed. Conversely, prototypes that can be exercised on a more or less *continuous* basis do not depend on complete development of a specific version of the system.

Revolutionary versus Evolutionary Prototyping.

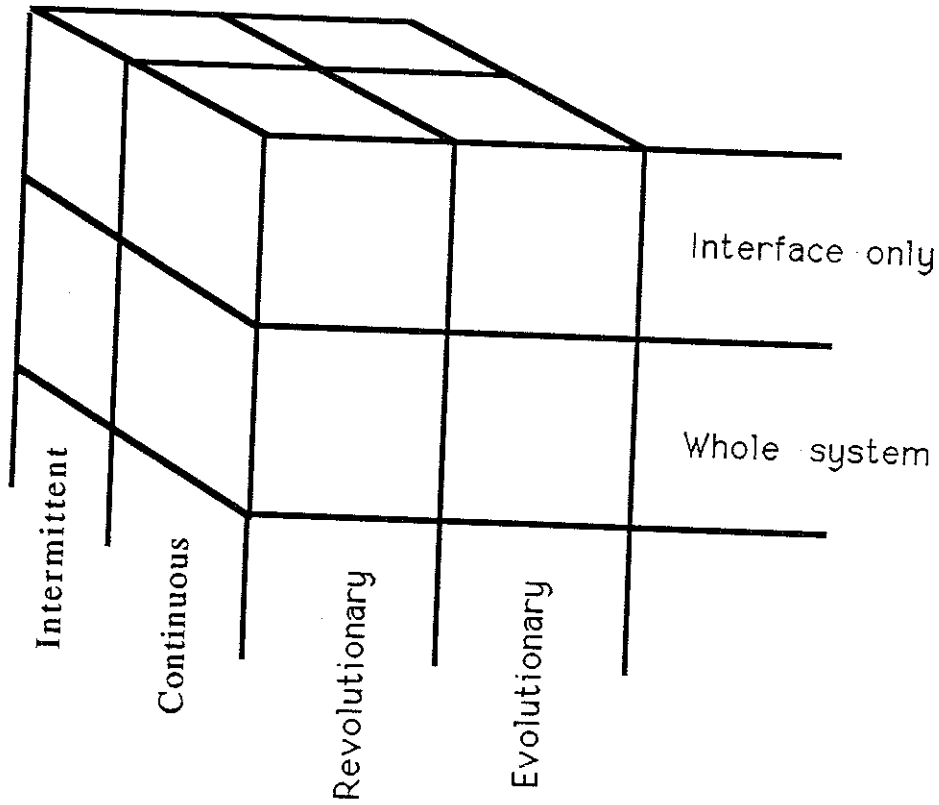

Figure 1. *Dimensions for classifying kinds of prototypes.*

It is not unusual for a software product to pass through several incarnations. The following steps are common:

1. one or more prototypes

2. a development implementation

3. the final product

The scope of this chapter includes steps 1 and 2, from which the first version that

can be called "the product" appears. The process of going from step 2 to step 3 is a "software manufacturing" step, applied only to a system that is fully developed. In the third step implementation is streamlined and optimized by "code-smiths," often into assembly language. This step is justified only if the potential market is large enough to amortize the effort or if there are special requirements for storage space, performance, or reliability (e.g., in a Department of Defense or NASA contract). A software company with a modest commercial market will usually sell the development implementation as the final product. The product itself may also be evaluated, possibly by the market, leading to a new version of the product. In such a case, the earlier product was a kind of prototype, too. If step 3 was involved in the development, there is a danger that, when modifications are needed, programmers may attempt to make changes directly to optimized code. This, however, can cause a loss in project management control and documentation, not to mention a gross deviation from development methodology. The software manufacturing step is never considered as part of the prototyping process; in the long run it is usually easier and more effective to change the development representation and regenerate the optimized version. This is especially true if there are automated tools to help with the optimization process.

The prototype-based development process is *revolutionary* if the prototypes of step 1 are discarded in the process of going to step 2. In an *evolutionary* process, the step 1

prototype eventually becomes complete enough to be a step 2 implementation. The nature of the evolution to step 2 depends on representation of the design in the prototype. If the prototype is coded, it may just be a matter of cleaning up the code and adding computational functionality. If the interface is represented in other ways (e.g., state diagrams representing dialogue control), implementation can be achieved by manually coding the state diagrams or, if suitable tools are available, through compilation of the representation that previously was interpreted in the prototype. A revolutionary prototype is most useful when it is built as early as possible and as rapidly as possible, without a large commitment of resources. Otherwise, deadline pressures make it difficult for managers and developers to work on a large prototype they know will be discarded. An early switch from a revolutionary prototype, however, means that development at the end, when changes can be surprisingly large and frequent, is done without benefit of a prototype. On the other hand, a revolutionary prototype can seduce developers into the trap of overdesign (Mantei, 1986). It is possible to become too attached to a prototype and invest too much in its development, only to have it scrapped. The engineering maxim of "making it good enough" applies particularly to throw-away prototypes. Perhaps the best way to avoid most of these problems is to adopt the evolutionary approach and not have to face the question of when to discard the prototype.

Interface Only versus Whole System Prototyping.

An interface only kind of prototype is sometimes called a façade or dialogue simulation, and the drawbacks are obvious. Dialogue situations dependent on computational actions can be difficult to anticipate in the interface. For example, the complicated dynamics of formatting displays for paging and scrolling of retrieved database records within a window are difficult to design without some real output for testing. Also, the dialogue developer cannot provide realistic messages in response to computational conditions not fully known or understood. If the computational component cannot be tested with the interface prototype, it is more difficult to integrate the interface design with the rest of the software. As computational functions come into existence, it is greatly beneficial to be able to see them in action in the prototype. Finally, of course, a prototype cannot be fully evolutionary unless the whole system is included.

Intermittent versus Continuous Prototyping.

One of the most common kinds of prototype is "implementation as prototype". The idea is to implement a "bread-board" mockup of the system to observe its behavior. Because the prototype is coded in a programming language, it is an effective way

to construct a whole system prototype. The disadvantage is that there are only *intermittent* times when the system representation (i.e., the code) is in a state that can be executed and evaluated. There are long intervals when, due to incomplete implementation of routines, syntax and semantic coding errors, data typing problems, unresolved symbolic references, and so on, it cannot run. Anything syntactically incomplete or erroneous in the partially developed code will prevent the prototype from executing. Configuration management, which reverts to the most recent complete version, does not help, because partially developed modifications occuring since the previous version are what need immediate testing. The result is *slow prototyping,* not a process that is useful for evaluating many different alternatives in an interface design. A secondary negative effect is batching of modifications to be made. Since there are only particular times when all routines can run together, large and small changes tend to get lumped together for the next version of the prototype. Every modification to a version must then take as long as the longest item and results of any changes are not seen until the next whole version is ready. As a result, the large number of small iterations required for such design decisions as syntax, message wording, and sequencing take a long time to stabilize. For example, the small modifications that can be involved in consistent assignment of programmed function keys to commands over an entire interface require testing of several configurations because each one is a compromise involving many screens throughout the interface. A slow batch-oriented development process does not serve this need. Rapid prototyping allows the designer to concentrate

on coherent treatment of such a problem directly and get it under control early on, rather than having to mix it in with all other interface problems. It is useful to cycle through the life cycle phases of design and evaluation for one or two interface features independently of the rest of the design. The software development principle of continuous evaluation (Boehm 1983) is to be taken quite literally in the realm of end-user interface development.

For most applications an evolutionary, whole system, continuous prototype is a desirable choice for the human factors developer. However revolutionary, interface only, intermittent prototypes are much easier for the computer scientist to provide mainly because most programming environments require programs to be complete and correct. The section on "The Problem of Prototyping Incomplete Designs" discusses this problem in more detail.

SYSTEM DEVELOPMENT ISSUES

Methodology and Life Cycle.

Three major factors are essential to a development methodology using rapid prototyping:

- iterative development using a high degree of iteration (especially early in the life cycle),
- end-user involvement, and
- *rapid* (versus slow) prototyping.

As pointed out earlier in the section on "The Concept of Prototyping", there is a difference between a development approach based on iteration, even with prototyping, and one based on *rapid* prototyping. Iterative development can be based on intermittent prototyping or it can be used without prototypes, simply by producing successive versions of the product. The important aspect of these approaches is that, although they are iterative, they are linear in the sense that they tend to go through the entire life cycle in a large loop. (This causes the batching problem mentioned in the section on "Intermittent versus Continuous Prototyping".) Effective development, especially if interface quality is an important factor, requires a process that is responsive to iteration

needs at a much finer resolution. Occasionally this can necessitate several cycles of redesign and evaluation just for a single interface feature. Rapid prototyping provides the means by which such local iteration can be accomplished. Not every approach that uses the term prototyping, then, is an example of the topic of this chapter (e.g., Bally, Brittan, and Wagner 1977).

In seeking a suitable development methodology, no place is found in traditional top-down, stepwise decomposition methodological models at which rapid prototyping concepts can be integrated (Mantei, 1986). A new life cycle is needed, one that will be a departure from the traditional approach in at least three ways:

- it will be a process of real cycling and less a linear process,

- developers can enter the cycle almost anywhere, and

- initial development proceeds bottom-up, starting with concrete design details of a scenario-based prototype (after beginning with task analysis and requirements specification).

This combination implies a high degree of localized cycling for individual interface features. It also implies that the maxim of "specifications always before design" no longer applies. The developmental approach mentioned earlier in "A Natural Technique" begins with a concrete example design (as a rapid prototype), which then feeds back to the more abstract requirements statement and specifications. This is exactly what happens in scenario-driven design. A scenario is specific about *how* interface features are designed. From this initial design the specifications of *what* is to be done can be deduced. Then once more the process moves forward in the cycle to refine the design, or perhaps to change the design significantly, and so on back to tuning the specifications.

In case studies done in the Dialogue Management Project (Hartson and Hix, 1986), it was observed that, overlaid upon local cycling and phases of the life cycle, there is an interesting progression by developers through various levels of abstraction. Developer activity throughout the overall interface development cycle appears to be a series of alternating waves of upward (bottom-up) and downward (top-down) progressions. It was observed that less experienced as well as more experienced developer subjects, when not constrained by a particular methodology, often started with interface scenarios — sketches of screen sequences as seen by the user. This is a concrete, bottom-up

approach and a natural way to begin, according to developmental theory. Some successful end-user testing was accomplished even with these very early tentative scenarios. Many developer subjects then produced a state diagram or similar flow chart-like representation to show a more general view of transitional relationships among interface screens. In additional bottom-up development, a control structure was designed to reflect state diagram information. At some point, developer subjects worked back downward through the now emerging levels of abstraction to analyze, organize, and modify the design. During this downward pass, for example, developer subjects faced issues such as consistency, grouping functions by similarity, and sharing and re-using functions. Then, typically, more end-user testing entered in, using a prototype interface. Similar observations about natural interactive system development were made independently by Carey (1987).

There appear to be essential differences in the nature of upward and downward movements during the interface development process. Upward development activities are typically empirical, synthetic, and behavioral; moving from abstract to concrete, they tend to reflect the end-user's view. Downward activities are more theoretical, analytic, and structural; moving from abstract to concrete, they tend to reflect the system designer's view. As a very simple illustration early requirements specification might be bottom-up, starting with details. Top-down hierarchical task analysis follows,

leading to a bottom-up scenario design. A control structure to support the scenario is developed top-down, returning to bottom-up end-user testing of details, followed by top-down modeling and abstraction to restructure the design, and so on.

Because the influence of the behavioral scientist is finally, and rightfully, becoming a factor in development of interactive systems, a final cautionary note is warranted. Some methodologies for developing interfaces are beginning to emerge from the behavioral and human factors side of the discipline without concern for connections to the software development process. There is a great amount of non-interface software with which the interface must be integrated in an interactive system and methodologies without connections to this software cannot be considered as serious possibilities to meet real world development needs.

Tools and User Interface Management Systems.

Although prototyping, and to some degree even rapid prototyping, can be accomplished without the aid of automated support, management problems can quickly become intractable. Use of computer aids in constructing and documenting designs and

prototypes can help a great deal if the tools used are well designed. Tools can maintain information about configurations, various versions, and reasons for design changes. This information, gathered as tools are used (both by developers and by end-users), enables the manager to track the fast pace of change during the design and prototyping phase of development. Automated tools can also provide metering necessary to obtain objective measures for prototype performance.

From the interface developer's viewpoint, the important role of automated tools in rapid prototyping is to support the highly iterative cycles of design and evaluation. In the context of end-user interfaces, User Interface Management Systems (UIMS) are becoming a key tool in this capacity. Modern UIMS, some of which were described briefly in the related work section, allow very rapid definition and alteration of the interface portion of a prototype. Many include simulators, or dialogue definition interpreters, which allow the interface to be designed and prototyped entirely within the UIMS. Automated tools are almost essential for using the rapid prototyping approach to system design. A more thorough discussion of the technical aspects of tools for rapid prototyping follows in the section on "Technical Problems and Solutions".

Design Evaluation.

Given that the computer science role can provide its partner, the behavioral science role, with rapid prototyping, the question is: What will the behavioral scientists do with it?

Traditional Controlled Experimentation. A conventional controlled experiment for point testing begins by stating a hypothesis of what is being tested and the expected outcome. For example, the hypothesis might be that "on-line help information is more effective than hard-copy manuals". An experiment is designed to test the hypothesis, beginning with a task for human subjects to perform. Independent variables are identified, often involving a single feature (such as the form of help information) to be tested. Dependent variables are objective measurements of end-user performance (e.g., task completion time, error rates). Data are collected, analyzed, and the hypothesis is confirmed or refuted. Occasionally, results of one or more experiments can be extrapolated into an interface design guideline or principle. Further experimentation can then be used to validate the principle. This is a gross simplification of the controlled testing process, but sufficient for this context.

This kind of testing is an important research tool that contributes to our store of knowledge. In time such basic empirical knowledge of human performance is translated into guiding precepts that are to be interpreted within specific design situations. The process of testing individual interface points and features, however, is not the effective evaluation process needed as a keystone within the interface development cycle. A large system simply cannot be decomposed into testable variables. It is not possible to isolate all the factors that affect usability, and not time or other resources to test them, anyway. Furthermore, testing all the parts is not the same as testing the whole integrated system. Short, and perhaps less formal, experiments can still be used to decide among alternatives for a given interface feature. However, a different approach to testing is needed to drive the iterative refinement process of fitting the system to the user; this approach must treat the target system, or at least its interface, as a whole.

Holistic Testing. Whiteside and Wixon (1985) view system testing from the perspective of psychological theory. *Behavioral theory* is presently dominant in system design and evaluation. The behavioral view is a mechanistic view focusing on cause and effect of isolated phenomena. Human behavior is shaped by the environment, as a passive reaction to the stimulation of reward and punishment. Within the context of human-computer interaction, this leads to adaptation of the end-user to fit the system! The end-user's behavior is shaped by error messages, feedback, and help information.

In contrast *developmental theory* takes an organicist view, that the human is a living and changing organism, too complex to impute cause and effect, especially to isolated phenomena. Human behavior is rational and rule-guided; knowledge is acquired through action. Emphasis is on studying behavior over performance, yielding more insight into reasons why end-user performance is bad or good in order to *change the system* to fit the user. The developmental approach, in the iterative development cycle, is amenable to observation, intervention, manipulation of conditions, and hunch testing. Controlled experimentation has a very narrow focus; developmental research gladly trades precision for breadth of scope. Developmental testing is holistic, including whole systems and their contexts, seeking "ecological validity" (Whiteside and Wixon, 1985).

Evaluation with Rapid Prototypes. A thorough coverage of user-based testing for the evaluation of system and interface designs is well beyond the scope of this chapter. [Editor: Reference to appropriate chapter?] However, evaluation is an important part of the iterative refinement process, and that process is tied closely to rapid prototyping as a development approach. It, therefore, is reasonable to focus briefly upon the subject of evaluation in relationship to rapid prototyping.

Use of rapid prototypes is an excellent way to approach holistic testing. It is essentially the only way to achieve early testing of whole system, or at least whole interface, designs. Because prototypes are often developed bottom-up, from interface scenarios, the method is very compatible with the developmental psychological view. In contrast to point testing discussed in the section on "Traditional Controlled Experimentation", a kind of evaluation that treats the whole system is a *case study* of subjects using the target system prototype to perform a task. Techniques described here are applicable to very early paper and pencil prototypes as well as computer-based prototypes used throughout most of the rest of the development cycle. Experimental sessions should be videotaped so they can be replayed as needed for analysis. One important technique for extracting information from the end-user is *verbal protocol taking.* In this method the subject is asked to discuss, by thinking aloud, the approach taken, problems experienced, and needs arising during performance of the task. Verbal protocol methods add to a case study by revealing thought processes behind observable events.

Perhaps the most useful technique for use with rapid prototypes in a case study evaluation is the *critical incident.* This technique is based on distinguishing situations and events, occurring during experimental observation, that significantly influence (either positively or negatively) performance of the task by the subject. The critical incident technique adds to the case study approach by distinguishing significant data

from the noise and bulk of the total data, and there is documented evidence of its validity and reliability (Andersson and Nilsson, 1964). Suitability of the technique when used with rapid prototypes is underscored by the fact that it is one of the few evaluation techniques that is effective for translating results into feedback of redesign requirements (Dzida, 1978).

Post-session interviews and *questionnaires* are also effective ways of extracting more information from subjects, especially if questions are well designed to lead to new interface requirements or design modifications. Open-ended questions can also be useful for getting at subjects' thoughts on what parts of the interface need improvement and why. Examples of rather successful application of the case study approach to user-feedback-driven development can be seen in a small number of commercial products today (Smith, Irby, Kimball, and Verplank, 1982; Tesler, 1983).

The prototype itself can capture end-user feedback, too. Each screen of the interface can be augmented with an additional end-user option, referred to generically as a "complaint button" (a slightly less euphemistic and more alliterative phrase is often used). When exercising a rapid prototype, the end-user will have one extra menu selection, PF-key, icon, or command on each screen for posting complaints or praise about features of a new interface/system. The complaint button is an especially good

way to get feedback about smaller details that may be forgotten before the post-session interview. It allows the end-user to express his or her feelings at the moment they are experienced. Even a short delay can diffuse or defocus those feelings. Some problems are a problem to a end-user for only a short time. After that, the marvelously flexible human may adapt and smooth over the rough spots interface developers are trying to detect through the refinement process. The naive user, as an evaluator, is a precious and perishable commodity.

A UIMS is an ideal tool for automatic insertion of features such as the complaint button into the prototype interface. Interface development tools can also be made automatically to build instrumentation into the prototype for monitoring end-user performance. Metering can add information about the internal state of the interface or target system, information essential for correlating redesign of system structure with new requirements revealed by testing. Detailed metering can provide an ability to associate performance times and error rates with specific features and parts of the interface. These kinds of data can allow developers to pinpoint parts of the interface that cause delays or errors in performance of the end-user's task.

TECHNICAL PROBLEMS AND SOLUTIONS

The Problem of Prototyping Incomplete Designs.

Because a prototyping approach to interface development allows for earlier error detection, errors are often easier and less costly to correct (Boehm, 1976). Thus, the ability to observe behavior of partially specified, partially designed interfaces (and systems) is of great value in their development. This is true, also, whether the partial design is mainly top-down or bottom-up in its development. Weighed against this payoff, however, is the fact that the early part of the life cycle is where technical problems with prototyping are greatest. When the design is less well developed, it is more difficult to "execute" a prototype. The difficulty stems from a simple fact: Computer programs are fragile. The slightest change to a program, the slightest error of commission or omission, can prevent it from running. Systems of software are even more fragile. All of a system's routines must each be "perfect" and so must all the interconnections and interrelationships represented by parameters and arguments, symbolic names, and data types. In contrast, prototypes, especially early ones, are characterized by incompleteness, ambiguity, tentativeness, and errors. These characteristics are all the opposite of what programs need to run.

While stubs can be used for routines not yet implemented, a stubbed system must still be syntactically complete and correct to compile and run, or even to be interpreted. This is a major drawback with interfaces that are programmed, either in a programming language or a high level dialogue language. Thus, prototyping calls for a support environment radically different from the traditional programming environment. A partial prototype must not just quit running when it does not have everything it needs for execution. In particular, life support mechanisms are needed to keep the software, especially that of the interface, alive until its ill-formed and damaged limbs and organs can be molded into single correct and complete design.

Allowing syntactically, as well as semantically, incomplete designs to be effectively executed as prototypes is one of the difficult technical problems in providing a usable rapid prototyping facility. Solutions to this problem are a serious challenge to the computer science role, but may not be appreciated as such by the behavioral scientist.

The Information Management Problem.

During design and prototyping phases of system development, an enormous amount of information is produced. Because the rapid prototyping approach to inter-

face development further introduces many designs and variations for the same system in an environment in which more than one development phase can be active concurrently, the problem of keeping track of the documents, design decisions, and personnel assignments is multiplied. In short, information that managers are accustomed to having could be more difficult to obtain, use, and maintain.

The problem of information loss becomes an important consideraton when using rapid prototyping. A long standing problem with software production has been the amount of information lost between phases of the software life cycle (Balzer, Cheatham, Green, 1983). The reasoning behind a given design or design change and the history of revisions are often not available to implementors and almost never available to maintainers (Gutz, Wasserman, and Spier, 1981). Often this kind of information goes completely unrecorded. With rapid prototyping, the problem becomes more pronounced, as this kind of information may be lost at each iteration. The result can be a lack of control of the process, and in some cases even a wasteful repetition of thought processes and previously rejected designs.

Solutions to the problem of managing the information produced and consumed throughout the development process, in particular during design and prototyping phases, involve two major components. One component is use of *automated tools* to

interactively create and record designs and prototype descriptions. Products of these tools are usually in the form of data and procedures that define target system objects and operations. More sophisticated tools may actively support tracking and documentation of designs, changes, and developer products and responsibilities.

The second component of a solution to the information management problem involves use of a *common database* among all tools used on a particular project, to manage all information produced by design and programming teams in a uniform and integrated manner. Requirements, specifications, scenarios, state diagrams, design notes, structure diagrams, memos, management information, and even source code for the entire project would reside in this database. This would provide a single, on-line repository for all information relevant to the project (Smith, 1986). Tool-to-tool communication of design representations can be enhanced by means of views (Date, 1985) tailored to provide information in a form suitable for each tool. These views would map information produced by each tool to a single canonical schema. Thus, tools can share information through a common database and yet maintain the representation which best suits their needs.

The Programming Problem.

There are technical problems with a programming approach to dialogue in a rapid prototyping environment. First, the dialogue developer is often considered to be a non-programming role. Also, as mentioned earlier, using stubbed programs or high level dialogue languages to prototype systems can cause difficulties, since programs must be syntactically complete and correct. This is especially problematic within the dialogue component of a new system. For a manually programmed interface, it is difficult to present alternatives quickly to end-users in order to evaluate usability. The difficulty of altering the interface prototype design becomes a major factor affecting the time between iterations of the prototype. When each iteration is a fairly lengthy process of programming and debugging, fewer iterations will be possible and end-users will have correspondingly less opportunity to participate in the design process. The process becomes more lengthy if large numbers of procedures and/or modules must be relinked, even after the slightest change is made. To some extent this delay can be controlled (e.g., by dynamic linking in an interpretive LISP environment), but it is still a case of programming.

On the other hand, tools such as UIMS allow many alternative designs for an interface to be tried in a shorter period of time (i.e., hours as opposed to days or

weeks). Relatively recent UIMSs allow rapid definition and demonstration of interfaces for a wide variety of types of systems. Though many of them concentrate on particular interaction styles or are limited to certain types of application systems, they are still useful tools for trying out initial ideas with users. Finer points not addressed by UIMS can be added in later prototypes or in the final implementation.

Since many UIMSs employ dialogue design languages which are menu-driven, form-based, or supported by semantically tailored editors, it is much easier to specify an interface that is syntactically valid. Also, errors in semantics are reduced by allowing the designer to view or execute the interface as it is being designed. Because of the ease with which they permit end-user interfaces to be defined and changed, UIMS may prove to be the single most important class of tools in decreasing the design/prototype iteration time.

The Communication Problem.

Rapid prototyping serves as a technique to communicate more effectively with users. Because of the increased rate of change of early system descriptions, it also demands more effective communication among developers. So far, automated tools

such as electronic mail systems have been inadequate to solve the problem. This may be largely due to the rather limited nature of these tools. How does one send a design graph through a conventional electronic mail system? How about interface screens? Performance evaluation charts? How does one find and access non-textual design documents and specifications referred to in a textual electronic mail message?

To enhance communication during design activities, a set of on-line communication aids must be integrated into the project design and documentation tools, the common database, and the library of target system source code. This would allow developers to embed messages with references to non-textual documentation that can be materialized immediately in a graphical form. Such a set of communication tools would allow for much more effective interaction among various developer roles.

An Environment for Rapid Prototyping.

A difficulty with some automated design tools is their lack of communication with each other, i.e., the lack of composability of their products. For example, output of tool A may be unacceptable as input for tool B. Tools may make use of different information representations because they were produced by different manufacturers.

Another problem with these tools is that they must run under conventional operating systems. Tool developers often find themselves restricted in the power they can provide because they can only assume a minimal amount of host system support. Most currently popular operating systems are designed to be general purpose environments for development, maintenance, documentation, and execution of systems of all types. Thus, designers of an operating system must try to make it a compromise between efficiency and power in all of these areas, with efficiency and high performance at execution time emphasized almost universally. A class of operating systems, referred to as development environments, is needed to deal specifically with problems of interactive system design and development. Only development tools themselves, and not target systems under development, are required to run fast and efficiently in this development environment, giving significantly different weight to considerations involved in operating system design. Since many problems of execution-time efficiency are less pressing, additional design-time power (e.g., dynamic linking, interpreters, debuggers) can be given to the operating system so that it may better support design and development tools, and thus the software development process.

As an example, in the context of rapid prototyping, consider the problems facing the designer of a tool to interpret executable design representations. The interpreter should be able to pass control, as needed, to already compiled parts of the prototype.

This feature would be particularly useful for attaching computational semantics to an interface only prototype. However, to accomplish this on most current systems, either the developer's code must be statically linked to the interpreter each time the prototype is changed, or the tool developer must create a new dynamic linking facility for each system to which the tool is ported. The first option is time consuming and increases the time between prototype versions. The second involves a complex programming task that would substantially increase the cost of the tool.

Use of a common static linker, to resolve all external references before execution, is justified only when all such references are known at link time and when the program will be run many times between changes. The former is not true in the case of rapid prototyping and the latter is almost never the case in a development environment. It is desirable for a development environment to incorporate a dynamic linking facility to be used by tool and software developers to make prototyping easier and faster.

There are other useful facilities in a development environment based on rapid prototyping, which are so computationally expensive that most conventional operating systems do not provide them. One of these is a facility for the maintenance of persistent objects. Object-oriented programming (Cox, 1986) is being increasingly used, even for developing the end-user interface. Most popular operating systems provide no support

for implementing and using objects. (Readers who are not familiar with object-oriented programming are referred to other sources: Cox 1986; Goldberg and Robson 1983; Goldberg 1984.)

CONCLUSION AND FUTURE

Rapid prototyping is a relatively new concept in the software industry. As it increases in popularity and maturity, many changes will occur in how interactive systems are constructed. Increasingly powerful, automated tools will become available. These tools will allow more rapid production of executable designs or specifications. Improved development environments and other tool packages will allow better coordination of multiple designers. Development, management, and communication tools will become better integrated. The overall result will be faster iteration through the design/prototype loop leading to systems that are faster and less expensive to produce and more satisfactory to end-users.

As more experience in application of rapid prototyping techniques is gained, new methodologies will remedy the shortcomings of currently popular software development methodologies by emphasizing the human-computer interface and by specifically supporting use of prototypes. These methodologies will define developer and end-user roles

for human-computer interface design and evaluation, roles lacking in current methodologies. They will explicitly include iterative refinement in the life cycle.

The future promises improved tools set in better software development environments, together with rapid prototyping methodologies and a body of practical experience. These are already combining to revolutionize interactive system development, and especially development of the human-computer interface.

ACKNOWLEDGEMENT

The authors wish to thank Pat Cooper for her cheerful and enthusiastic care and typing in the face of severe and unmerciful hardware difficulties. Special appreciation is expressed to Deborah Hix for her careful reading of the manuscript and her comments and suggestions for improvement. Thanks for help also go to Jim Callan, Beverly Williges, and Roger Ehrich. We also gratefully acknowledge research support, under the supervision of Dr. H. E. Bamford, from the National Science Foundation; under the supervision of Mr. T. M. Kraly, from IBM Federal Systems Division; and from the Virginia Center for Innovative Technology.

REFERENCES

Alavi, M. (1984) An Assessment of the Prototyping Approach to Information Systems Development. *Communications of the ACM, 27*,6, 556-563.

Andersson, B. and Nilsson, S. (1964) Studies in the reliability and validity of the critical incident technique. *Journal of Applied Psychology, 48,* 6, 398-403.

Bally, L., Brittan, J., and Wagner, K.H. (1977) A Prototype Approach to Information System Design and Development. *Information and Management, 1,* 1, 21-26.

Balzer, R.M., Cheatham, T.E., and Green, C. (1983) Software Technology in the 1990's Using a New Paradigm. *IEEE Computer, 16,* 11, 39-45.

Boehm, B. W. (1976) Software Engineering. *IEEE Transactions on Computers, 25,* 12, 1226-1241.

Boehm, B. W. (1983) Seven Basic Principles of Software Engineering. *The Journal of Systems and Software, 3,* 3-24.

Boehm, B. W., Gray, T.E., and Seewaldt, T. (1984) Prototyping Versus Specifying: A Multiproject Experiment. *IEEE Transactions on Software Engineering, 10,* 3, 290-303.

Callan, J. E. (1985) Behavioral Demonstrations: An Approach to Rapid Prototyping and Requirements Execution. Unpublished masters thesis, Virginia Tech Department of Computer Science, Blacksburg, VA.

Carey, T. (1987) The Gift of Good Design Tools. To appear in H. Rex Hartson and Deborah Hix (Eds.), *Advances in Human-Computer Interaction,* Vol. 2. Norwood, NJ: Ablex.

Carroll, J. M., and Rosson, M. B. (1985) Usability Specifications as a Tool in Iterative Development. Chapter one of H. Rex Hartson (Ed.), *Advances in Human Computer Interaction, Vol. 1,* (pp. 1-28). Norwood, NJ: Ablex.

CBS, Inc. (1986) Interview with Anthony Perkins. *CBS Morning News,* Monday, 7 July.

Cox, B.J. (1986) *Object Oriented Programming: An Evolutionary Approach.* Reading, Mass.: Addison Wesley.

Date, C.J. (1986) *An Introduction to Database Systems, 4th Ed.* Reading, Mass.: Addison Wesley.

Dzida, W., Herda, S., and Itzfeldt, W.D. (1978). User-perceived quality of interactive system. *IEEE Transactions on Software Engineering, SE-4,* 4, 270-276.

Flanagan, D., Lenorovitz, D., Stanke, E., and Stocker, F. (1985) *RIPL Concept of Operations and System Architecture.* CTA Internal paper. Englewood, Col.:Computer Technology Associates.

Goldberg, A., and Robson, D. (1983) *Smalltalk-80: The Language and Its Implementation.* Reading, Mass.: Addison Wesley.

Goldberg, A. (1984) *Smalltalk-80: The Interactive Programming Environment.* Reading, Mass.: Addison Wesley.

Gutz, S., Wasserman, A.I., and Spier, M.J. (1981) Personal Development Systems for the Professional Programmer. *IEEE Computer, 14,* 4, 45-53.

Hanau, P. R., and Lenorovitz, D. R. (1980) A Prototyping and Simulation Approach to Interactive Computer System Design. In *Proceedings of the Design Automation Conference,* ACM.

Hanau, P., and Lenorovitz, D. (1980) Prototyping and Simulation Tools for User/-Computer Dialogue Design. In *SIGGRAPH Proceedings* (pp. 271-278). Seattle, Wash.: ACM SIGGRAPH.

Hartson, H. Rex. (1985) Preface to *Advances in Human-Computer Interaction,* Vol. I, Norwood, NJ: Ablex.

Hartson, H. R., and Hix, D. H. (1986) UIMS: Toward the Next Generation, Technical Report TR-86-41, Department of Computer Science, Virginia Tech, Blacksburg, Va. 24061

Mantei, M. (1986) Techniques for Incorporating Human Factors in the Software Lifecycle. In *Proceedings Structured Techniques Association Third Annual Conference* (pp. 177-203). Chicago, IL.

Mason, R. E. A., and Carey, T. T. (1981) Productivity Experiences with a Scenario Tool. In *Proceedings of COMPCOM '81* (pp. 106-111). Washington, D.C.: IEEE.

Mason, R. E. A., and Carey, T. T. (1983) Prototyping Interactive Information Systems. *Communications of the ACM, 26,* 5, 347-354.

McFarland, G. (1986) The Benefits of Bottom-Up Design. *ACM SIGSOFT Software Engineering Notes, 11,* 5, 43-51.

Piaget, J. (1952). *The Origins of Intelligence in Children.*New York: International Universities Press.

Smith, D.C., Irby, C.I., Kimball, R., and Verplank, W. (1982) Designing the Star User Interface. *Byte* (April), 242-282.

Smith, Eric C. (1986) System Support for Design and Development Environments. Masters thesis, Department of Computer Science, Virginia Tech, Blacksburg, Va.

Tesler, L. (1983) Enlisting User Help in Software Design. *ACM SIGCHI Bulletin 14,* 3, 5-9.

Wasserman, A.I., Pircher, P. A., Shewmake, D. T., and Kersten, M. L. (1986) Developing Interactive Information Systems with the User Software Engineering Methodology. *IEEE Transactions on Software Engineering, 12,*2, 326-345.

Wasserman, A.I., and Shewmake, D.T. (1982) Rapid Prototyping of Interactive Information Systems. *ACM SIGSOFT Software Engineering Notes, 7,* 6.

Wasserman, A.I., and Shewmake, D.T. (1985) The Role of Prototypes in the User Software Engineering (USE) Methodology. Chapter seven of H. Rex Hartson (Ed.), *Advances in Human-Computer Interaction,* Vol. 1 (pp. 191-209). Norwood, NJ: Ablex.

Whiteside, J., and Wixon, D. (1985) Developmental Theory as a Framework for Studying Human-Computer Interaction. Chapter two of H. Rex Hartson (Ed.), *Advances in Human-Computer Interaction,* Vol. 1 (pp. 29-48). Norwood, NJ: Ablex.

Wong, P.C.S., and Reid, E. R. (1982) FLAIR - User Interface Dialogue Design Tool. *SIGGRAPH Computer Graphics, 16,* 3, 87-98.