Rapid Prototyping in Human-Computer Interface Development

H. Rex Hartson and Eric C. Smith

TR 89-42

In revision for *Interacting with Computers*, Butterworth Publishing, U.K.

RAPID PROTOTYPING IN HUMAN-COMPUTER INTERFACE DEVELOPMENT

H. Rex Hartson
Eric C. Smith

Department of Computer Science
Virginia Tech
Blacksburg, VA 24061

October 1989
(Revised)

ABSTRACT

Some conventional approaches to interactive system development tend to force commitment to design detail without a means for visualizing the result until it is too late to make significant changes. Rapid prototyping and iterative system refinement, especially for the human interface, allow early observation of system behavior and opportunities for refinement in response to user feedback. The role of rapid prototyping for evaluation of interface designs is set in the system development life cycle. Advantages and pitfalls are weighed, and detailed examples are used to show the application of rapid prototyping in a real development project. Kinds of prototypes are classified according to how they can be used in the development process, and system development issues are presented. The future of rapid prototyping depends on solutions to technical problems that presently limit effectiveness of the technique in the context of present day software development environments.

Key words: rapid prototyping, human-computer interaction, prototyping, user interface, development environment, development methodology, evaluation, life cycle, software tools, user interface management

1. INTRODUCTION

1.1 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.

Automobile 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--tend to force 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 user dissatisfaction with many of the products so developed.

Recently, however, the techniques of prototyping, especially *rapid prototyping*, and iterative refinement have emerged in the context of software development, especially for the human-computer interface. Such techniques allow the software development process to share the essence of the Hitchcock story development scheme: refinement of the product based on feedback from users.

One interesting 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 development involves production of at least one early version of the system that illustrates 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 user interaction is much shorter. This, in turn, allows multiple iterations through the refinement process and a finer tuning to the needs of the user, leading to a high degree of confidence in the usability of the resulting system.

The technique of rapid prototyping may be applied to development of any part of a system. For quite some time there has been considerable interest in prototyping as part of the general software development process, without specific emphasis on the user interface [Tanik and Yeh 1989]. However, the focus in this paper is on the user interface and user-oriented issues such as learnability, usability, and functionality. 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 user can be "stubbed" in the prototype, saving time in its construction.

1.2 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 just a tool.* The technique can be effective even when performed manually, especially in the early, conceptual stages of development. Creative use of paper and pencil prototypes, flip charts, movable "stick-on" felt cutouts, and other props can sort out important aspects of an early design before any implementation effort is expended. One such prototype interface even included a borrowed telephone receiver for the user to seek help information. Members of the development team act out the role of the computer and evaluators act the part of the user. Typically, a great deal of discussion accompanies the interaction and weaknesses in the interface design are highlighted by the amount of extra-system dialogue required to clarify the meaning of various features and how to use them. Such theatrical dialectics seem to work best when oriented toward a specific task, user goal, and situational context. The interaction is concrete and fairly detailed. The results can often, however, be generalized to cover other parts of the design as well.

In fact, since rapid prototyping 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 it 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 users and developers, a prototype is concrete while specifications are more abstract.

Rapid prototyping is also essential to the notion of iterative refinement. It is not that developers 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 successive adjustments are made in order to hit the target.

Ehn [1989] views design as a cooperative process and describes communication among participants. Drawing on language philosophy of Wittgenstein, Ehn focuses on a shift from language as description to language as action. The prototyping approach is supported by his conclusion that some design requirements are best conveyed by *showing* rather than by just *saying*.

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. In the cooperative development activity of behavioral scientists and computer scientists, a gap exists between the skills and goals brought to the task by each of these roles [Hartson 1985]. Computer scientists often do not fully understand the need for user-centered design or the behavioral scientist's concern for human factors or how good human factors are achieved. Alternatively, the behavioral scientist often does not appreciate the limitations 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 analysis and evaluation. This is not just a temporary situation, either, until developers 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." Conversely, knowledge on the part of the designers that they do not have to get it right the first time offers greater opportunities for exploration and experimentation with a higher probability of a more innovative design. 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 development 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 conflict with those of the other role. Through rapid prototyping, an early opportunity is afforded the behavioral scientist to build good human factors into an interface design. By building ease of testing and 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.

1.3 Organization of This Paper

Section 2 weighs the advantages and pitfalls of prototyping, and section 3 introduces some dimensions from classifying various kinds of prototypes. Section 4 describes some experimental and commercial prototyping systems as examples to illustrate the classification scheme of section 3. Section 5 is devoted to an example of the application of prototyping to a real development project. In section 6, we look at the methodological and tool environment that surrounds prototyping in the development process, and some technical problems involved with prototyping are discussed in section 7.

2. WEIGHING RAPID PROTOTYPING

2.1 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 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 software development process (with no particular emhpasis on the interface). 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, possibly at a cost in generality of design. Finally, the prototyping groups accomplished their task with 45 percent less effort than the other groups.

In a similar study [Alavi 1984], users of systems developed using the prototyping approach were better disposed toward the product than were users of non-prototyped systems. Developers felt that prototyping enhanced communication about the 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." The prototype created a common baseline or reference point from which potential problems and opportunities could be identified. Discussions could take place between developers and 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 user acceptance of the systems. The first version that 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 facilitate earlier response to these changes 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 developers. 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. Often, 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, users can be presented with many options instead of a single design, increasing their ability to maneuver toward a design that 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.

2.2 *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 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 themselves be large. 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 section 6 on system development issues).

Some of the most serious problems occur if various parties begin to view the prototype as the final system. 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. The reader is referred to the subsection on management concerns under section 6.1 on methodology and life cycle for further discussion of management problems and solutions.

Prototypes with emphasis on the 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 almost always leads 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--such as with (rapid) prototyping--with the top-down step-wise decomposition process of the computational software of the design [Hartson and Hix 1989b]. 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.

Finally, as Weiser [1982] points out, a prototype is a scale-model of a real system and is limited regarding the accuracy with which it can represent the real system. Prototypes usually focus on one aspect of a target system--the user interface, system functionality, or system performance. If a prototype is accurate in one of these areas, it is unlikely to be accurate in either of the other two. Scaling back up to the real system will require attention to where these inaccuracies occur.

3. KINDS OF PROTOTYPES

There are a large number of techniques for prototyping. Gutierrez [1989] highlights the need for a model of problem types to help fit the techniques to the nature of the problem domain. Carey and Mason [1983] survey several specific techniques and tools used for large information system development efforts, along with a wealth of references to case studies.

Nielsen [1987] describes some dimensions on which a prototyping technique can be classified, based on how a target system is scaled down in the prototype. *Vertical prototyping* requires the number of features to be reduced, yielding a narrow system with a depth in functionality maintained. Evaluation can be realistic but only for a few functions. *Horizontal prototypes* offer less depth in functionality, but are kept broad in features, resulting in a shallow version of a full-featured system. Evaluation is less realistic but covers the whole system. A "scenario" is reduced in both dimensions, giving a quick, low-cost early prototype.

From the perspective of the development process there are at least four (more or less orthogonal) other dimensions along which approaches to prototyping can be classified. These dimensions are:

- specification: how are interface designs specified?
- maturation: how does the prototype grow into a product?
- scope: can prototype include the whole system or just the interface?
- executability: can prototype be executed at any time?

3.1 Specification: Techniques for Representing Interface Designs

Approaches to prototyping can be classified by the techniques used to specify interface designs. Prototyping tools share this need for interface specification with other interface development tools. Myers [1989] also uses specification techniques as a means for classifying interface development tools. A complete discussion of these techniques is beyond the scope of this paper, but is treated in Myers [1989] and Hartson and Hix [1989a].

Formal context-free grammars, expressed in Backus-Naur Form (BNF) are among the earliest ways of representing purely sequential control flow in dialogue. State transition diagrams (STDs), the graphical equivalent of BNF, are generally accepted to be easier to understand than BNF [Jacob 1982]. Jacob [1985] and Wasserman and Shewmake [1985] were among the leaders in using executable state diagram specifications to provide working prototypes. High-level "dialogue programming" languages have been used to represent both sequential (e.g., RAPID/USE [Wasserman and Shewmake 1985]) and asynchronous (e.g., CHISL [Gray, Kilgour, and Wood 1988]) dialogue.

Jacob has also adapted STDs for the specification of asynchronous (non sequential) interaction [Jacob 1986]. Event handlers [Green 1985; Hill 1987] are used to specify interface and system reactions to internal events that result directly from asynchronous user actions. In an interface tool called MIKE, Olsen [1986] uses an approach to generate an interface from a description of the system's schematic (computational) routines. The current trend is toward more graphical, direct manipulation specification techniques, and techniques that specify "by-demonstration" using inference and user interaction to establish general specifications from specific examples [Myers 1988].

The interface specification techniques mentioned above represent not so much a spectrum from which to choose as they do a historical progression. These techniques are evolving and improving with regard to usability for the developer. BNF represented early sequential dialogue structure. Graphical diagrams improved readability. The move to asynchronous, direct manipulation and by-demonstration interaction styles have prompted a move to similar means for representing interface designs. For interface developers who wish to concentrate on the interface and not the problem of its representation, the less the representation technique is like a programming language, the better. This is especially true as interface development is falling more and more to behavioral specialists rather than programmers [Richards, Boies, and Gould 1986].

3.2 Maturation: Revolutionary versus Evolutionary Prototyping

During its maturation from prototype to product, it is not unusual for a software system 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 paper 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 often sell the development implementation as the final product. 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 version 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* in its maturation if the prototypes of step 1 are discarded in the process of going to step 2. In *evolutionary* maturation, 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. In cases where hardware is involved, revolutionary software prototypes are often a necessity. To this end Virtual Prototypes, Inc. of Montreal provides techniques and tools to produce "virtual prototypes," software replicas of hardware environments for display and control (e.g., airplane cockpits) with automatically generated software and touch sensitive graphics. For interfaces where software design is the issue, evolutionary maturation is most useful when the prototype 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 to only implementation, 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. 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.

3.3 Scope: Interface-Only versus Whole-System Prototyping

When the scope of the prototype is limited to just the interface, the prototype is sometimes called a façade, interface simulation, or mock-up [Gregory 1984], and the drawbacks are obvious. Interface 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 and evaluate without some real output for testing. Also, the interface 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 in the scope.

3.4 Executability: Intermittently versus Continuously Executable Prototypes

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 the partially developed modifications occurring since the most recent running 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. When each iteration is a lengthy process of programming and debugging, fewer iterations are possible and users and evaluators have correspondingly less opportunity to participate in the design process. Also, of course, as the system grows, more and more delay is incurred from compiling, linking, and loading.

A secondary negative effect of an intermittent ability to execute a prototype 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 complete 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 interface developer to concentrate directly on coherent treatment of such a problem 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 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.

4. EXAMPLE SYSTEMS AND APPROACHES

4.1 Examples of Experimental Systems.

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 developers to construct useful prototypes while reducing the amount of conventional programming required. Starting in the 1970's and on through the 1980's several such environments and tools have been developed.

These have served as examples and starting points for much of the current research in rapid prototyping.

FLAIR. The Functional Language Articulated Interactive Resources (FLAIR) system was created at TRW to aid interface developers in involving users in the development process [Wong and Reid 1982]. FLAIR facilitates development 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 façades, making it interface-only in scope. A prototype requires considerable additional programming to "hard-wire" the behavior of the application system to give the appearance of whole-system behavior. FLAIR was among the first to provide a Dialogue Design Language as a dialogue specification technique. Rather than the (then) more common formal grammars, this language, with its voice-driven menu interface, is used to describe the user interface structure. A "show-by-example" menu method can also be used to specify dialogue.

Because of the amount of coding involved in a typical prototype, the prototypes tended to be only intermittently executable. Because prototypes were largely façades, they tended to be evolutionary (throw-away) and interface-only. The resulting prototypes, however, were very realistic and could exhibit complex graphical behavior.

IDS. The Interactive Dialogue Synthesizer (IDS) was developed in the 1970's at Martin Marietta as a tool to aid in the production of interfaces for command and control systems [Hanau and Lenorovitz 1980a; Hanau and Lenorovitz 1980b]. IDS is a good example of a system that uses Backus-Naur Form grammatical rules (with associated actions) as a technique to define interaction sequencing of the target system interface. 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 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.

IDS prototypes are interface-only, with dynamic sequences represented by timed sequences of static displays. As such they would be revolutionary (throw-away), but continuously executable. Like FLAIR, IDS has seen numerous large real-system applications.

RAPID/USE. RApid Prototypes of Interactive Dialogues (RAPID) is a tool designed for use 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 the sequencing of interaction. Displays are associated with state nodes and inputs with state transition arcs. Dialogue, the contents of the nodes, is specified with a high level textual dialogue definition language. Prototype interfaces can be defined and simulated rapidly with continuous executability. As the interface prototype becomes more stable, prototype application semantics may be attached using the Troll/USE database management package providing connections to whole-system prototypes and leading to an evolutionary maturation process. Eventually a fully operational prototype can be created. The USE methodology was 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. A product based on RAPID/USE is now commercially available from Interactive Development Environments, Inc.

Behavioral Demonstrator. The Behavioral Demonstrator [Hartson, Johnson, and Ehrich 1984; Callan 1985] is intended to support rapid prototyping within the Dialogue Management System. The Behavioral Demonstrator interprets designs specified with supervised flow diagrams, each a kind of state transition diagram that describes high level flow of control and data in a target system. Dialogue content is created using specialized direct manipulation tools. The interpreted nature throughout gives it the flexibility of a strong evolutionary approach, but causes it to suffer with slow performance. Computational functionality is either programmed or stubbed in, providing connections to whole-system prototypes. 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. Because of its continuous executability, systems and interface developers using the Behavioral Demonstrator can alter the design during the running of the prototype and restart from that same point in its execution, providing very rapid turnaround.

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 façade prototypes for complex interactive systems [Flanagan, Lenorovitz, Stanke, and Stocker 1985)] Interface components called "tiles" are created by the developer 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 advice to interface developers, and an "evaluation expert" is used to evaluate the prototype itself.

As in the case of FLAIR, considerable programming is involved in producing complex and realistic interface-only prototypes, making them revolutionary in their maturation and intermittently executable.

4.2 Commercial System Examples

New user interface tools are currently being announced at a prodigious rate and almost all such tools now have some kind of prototyping capability. It is impossible to mention them all here. This section is not intended to be a thorough survey or a "consumer's guide" to prototyping tools. It contains only representative examples selected to illustrate specific points.

ACT/1. ACT/1 [Mason and Carey 1981; Mason and Carey 1983], developed by Art Benjamin and Associates of Toronto, was one of the first commercially available products for rapidly prototyping user interface scenarios. ACT/1 employs a specification technique that allows developers to create interface screens by filling in table entries -- tabular forms of STDs -- on the screen. An advanced system for its time, ACT/1 is constrained to a narrow range of interaction styles (e.g., menus, forms, question and answer styles) compared to today's direct manipulation interfaces. Procedural links are specified in tabular form with entries having the format:

<input screen, process, output screen>

When the computational routines are coded, these process links can afford a whole-system prototype. Because it is not coded, the interface portion is continuously executable. At first, with no application logic specified, users may go through a fixed script simulation of the user interface. Application logic can be added to create a first prototype of a new system. ACT/1 can be used in an evolutionary approach to maturation; the prototype screens are directly usable in early production versions of target systems. ACT/1 has had more than one hundred users and has been applied to the development of several interactive information systems.

HyperCard. Because HyperCard comes essentially free of cost for use on Macintosh computers and is billed as "programming for the rest of us," it is widely known and used. Even though it is not intended just for prototyping, perhaps more prototypes have been built with HyperCard than with any other tool. Interfaces are specified in terms of "cards," interactive object-oriented screens. The contents and sequencing of the graphical and textual objects are specified by menus and direct manipulation graphics and text editors. Cards are grouped into stacks, which can share a common background, while the foreground changes from card to card. Cards have fields for text-entry and buttons, "hot spots" on either graphical or textual objects making the objects selectable by the end-user. Menus and scripts are used to specify what happens when a button is selected, giving the approach an event-based flavor.

The computational "actions" of small and medium-size applications can be programmed into "scripts" using the Hypertalk language. This ability to integrate application functionality, including small-scale database functions, plus X (external) function links to conventionally programmed routines provides a very broad whole-system scope. Since cards are interpreted, performance can be slow, but maturation is evolutionary and executability is continuous.

Many simple interface features can be produced easily and rapidly, yet more complex behavior is possible. The drawbacks are not serious. Form filling is not a well supported interaction style and, at present, HyperCard supports only a single window. A large and growing common library of user interface objects is provided to give the developer a running start.

SuperCard, the eagerly awaited HyperCard successor from Silicon Beach Software, supports multiple windows, has more features (e.g., animation), and is, in the main, compatible with (convertible from) HyperCard stacks. Interpreted execution still means slow performance.

Demo. The Demo program, designed by Dan Bricklin and grown by Software Gardens Inc., is one of the earliest commercially available prototypers for personal computers. Prototypes are based on a "slide show" concept, and as such, are interface-only in scope. The developer specifies the prototypes primarily by menu choices and text entry. Tabular mappings (a form of STDs) are used to link slides for sequencing. Early versions had very limited interaction styles (e.g., did not support use of a mouse). Also, no predefined high level interface constructs are supported. Users must construct even menus and forms from lower level elements, often with long sequences of menu selections.

The maturation process in revolutionary; a Demo prototype could not be used in a serious production version of a system. Executability is continuous.

Prototyper. The SmethersBarnes Prototyper system is a "user interface builder, simulator, and code generator" for Macintosh style interfaces. Its adherence to the Apple Human Interface Guidelines can be a blessing or a problem, depending on style needs. Interface element are specified by direct manipulation editors and drawing tools. Predefined higher level elements (e.g., icons, pictures, popup menus, lists) are provided along with a special editor for each. The "Quicklook" feature allows rapid execution of parts of the interface without leaving the tool or compiling code. The run-time performance penalty is avoided, however, by production of (well-commented) interface code in Pascal or C.

Prototyper is interface-only in scope. Unlike HyperCard, the tool itself involves only the interface. A non-programming interface developer cannot provide any functionality. (Whereas with SuperCard, for example, some functionality can be had with only limited programming skills.) Connections can be made from the generated code, however, to computation functions programmed in Pascal or C. The resulting collection of software is whole-system in scope. Because Prototyper generates interface code that can be linked to computational routines in a production version of a target system, the maturation is strongly evolutionary. Also, automatic generation of code (hopefully without errors) and the many demonstration modes (e.g., Quicklook) produce prototypes that are continuously executable.

Generally considered a positive feature, the generation of interface code can have pitfalls. Availability of the this code to the developer can pose a dangerous temptation. Changes can easily be made in the interface code without going back to the Prototyper tool. The code, however, immediately becomes unmaintainable via the tool. Therefore, all changes should be made with the tool, regenerating the code each time. The alternative is a return to conventional programming and a loss of this tool and its advantages.

Together with HyperCard, Prototyper is evidence of the increasing availability of powerful and flexible prototyping tools using direct manipulation specification techniques and approaching an evolutionary, whole-system, and continuously executable prototyping capability.

Transportable Applications Environment (TAE). TAE, developed by NASA Goddard Space Center in Maryland, may more properly be classified as a non-commercial user interface management system based on the X Window System, but it appears here because it has a strong prototyping component (as do most current UIMS), is currently available, and has large numbers of users at many (well over a hundred) beta sites. Specification is accomplished by a combination of techniques. Screen appearance is specified by a workbench graphics and text editing tool. More direct manipulation style could be employed in the workbench tool; often items are constructed in one place and then moved to the target system screen, rather than being constructed in final run-time form. This tool is supplemented with the TAE Command Language, a high-level dialogue programming language for expert interface developers. Some form filling is used, for example, to specify the computational routines for whole-system connections. The interpretable interface designs offer continuous executability and are robust and complete enough to allow evolutionary maturation.

Open Dialogue. Produced by Apollo as a successor to Domain Dialogue, Open Dialogue is an object-oriented tool layered on UNIX and the X Window System for building and prototyping user interfaces. Interfaces are specified by declarative definitions of the objects and their relationships. Interface elements can be specified at run-time, if desired, yielding continuous executability. Numerous mechanisms allow close coupling with the computational software giving a nearly whole-system scope. The computational side can also be stubbed for interface-only prototyping. For many applications the interface prototype is not a throw-away, thus maturation can be evolutionary.

5. EXAMPLES FROM A DEVELOPMENT PROJECT

5.1 HyperCard as a Prototyping Tool

User interface prototyping played an important role in the development of a recent version of the Dialogue Management System (DMS), a User Interface Management System (UIMS) being developed at Virginia Tech. We did our prototyping with HyperCard on a Macintosh II, also our machine for DMS implementation. While HyperCard is not intended as just a prototyping tool, we found it to be generally effective in that role during our development process. HyperCard allows very rapid screen mock-ups with built-in icons and some kinds of menus. Customized icons are simple to construct. The

HyperCard run-time environment contains a sophisticated input event handler. Sensitive screen areas, called buttons, can be defined to invoke arbitrary functions in response to picking by users. Because HyperCard is interpreted, programs (called scripts) can be run immediately after changes are made to them. The cost for this feature is slow performance and some difficulty in tracing errors. HyperCard functionality is extensible in the sense that any feature not already available (e.g., pop-up and pull-down menus) can be programmed (e.g., in C or Pascal) and installed as an X-function (external function).

Because we also used HyperCard to document our user interface designs, we were able to integrate prototyping and design recording tools in an interesting way. HyperCard buttons were associated with objects in screen pictures and other illustrations of the design documentation. If the reader of the on-line design documentation wishes to explore the use of a feature, a mouse selection of the corresponding button will transport the reader to the HyperCard prototype for that feature. We perceive great benefit for an even closer merging of the design documentation and prototype. Connections to a prototype can offer interactive documentation and training for the user of an application.

We found a need for two kinds of prototyping: global and local. The global prototype was used to allow early observation of general DMS behavior. Local prototypes were used to evaluate design alternatives for specific isolated interface details.

5.2 Global Prototyping

HyperCard was used for global prototyping of the DMS Graphical Programming Language (GPL) tool, a graphical editor for constructing and maintaining supervised flow diagrams. These diagrams are used to represent control and data flow in an interactive system design. The diagrams employ several different symbols for various kinds of nodes (e.g., entrances, exits, dialogue states, control states, computational states, and decision points) connected with arcs labelled with state transition conditions.

The DMS development team had distinct roles for interface developers, implementers, and evaluators. In the early design stages the primary user of the prototype was the interface developer, who used the prototype almost daily as feedback for trial-and-error iterative design during this time. During the first month or two of design, the prototype underwent major re-design about twice per week. The goal was to iteratively refine the design via the prototype, which we would later discard in favor of a Smalltalk implementation.

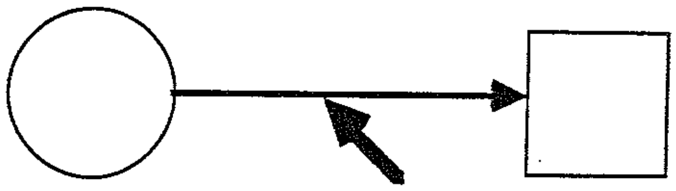
The GPL prototype was implemented by a newly hired programmer, who was unfamiliar with the Dialogue Management Project, HyperCard, and Smalltalk. He was shown a video tape of an earlier version of DMS and then began communicating with the lead developers about GPL. In two and a half weeks he had the basic functionality of GPL in the prototype, and required an additional week to polish it. During the ensuing iterations, most of the modifications were made by this programmer although the interface designer also had experience with HyperCard.

The main GPL screen is prototyped as a single card in HyperCard (see Figure 3) with buttons for each GPL icon and each supervised flow diagram object instance that is created. The modes (currently selected command or icon) are recorded in global variables. All mouse activity within the screen is sent to a "switch" qualified by the mode that is current. The appropriate script is then invoked for the user's task (e.g., drawing lines, moving symbols). Each supervised flow diagram developed by the user is kept on a separate HyperCard card.

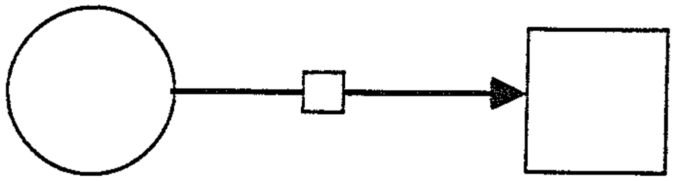
Out of a twelve month development process, the global GPL prototype had a useful life of about two months, at which point a major design review was held, followed soon by the delivery of a design document to our sponsor client. Beyond this point, the GPL prototype continued to serve the development process as a local prototyping tool for evaluating isolated design decisions.

Space here does not permit an extensive case study of the use of a prototype to probe cognitive issues, user models of a system, or formal user data gathering for design guidance in the iterative development cycle. It can be said, however, that having the prototype used by real end-users did help the designers see the interface from the user's viewpoint, exposing how users thought about the objects of the interface and their manipulation, relating them to task performance issues in the design. We can informally describe a few examples here to indicate the possibilities.

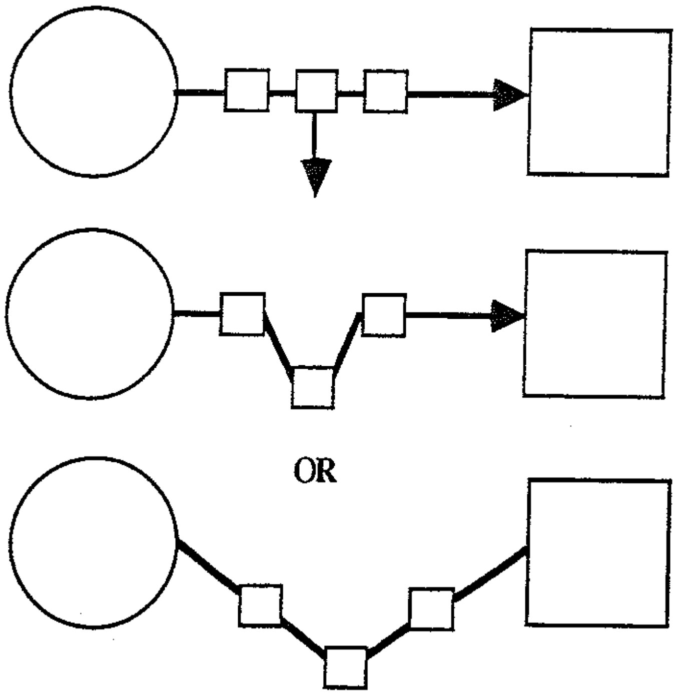
An example of a task-oriented issue studied with the prototype involved control flow arcs, which had several interdependent user-related problems. We wanted arcs to be responsive to direct manipulation by the user. Our initial designs had ambiguities and special cases, and did not address some important questions. A particularly difficult issue involved the means for selecting an arc. Must the user select the visible line precisely with the mouse (as in Figure 2a) or can we add "grab handles"? If so, should the handle be in the middle of an arc (as in Figure 2b) or one placed at each endpoint? Will grab handles add too much clutter in crowded diagrams? How should we represent a vertex in a segmented arc? How can vertices be made to articulate as joints? Does movement of a vertex imply movement of adjacent segments only, or of the entire arc (as in Figure 2c)? How can rubber-banding be used effectively? With the help of the prototype, the design here changed completely four times until a design that was acceptable to developers and a number of users was produced.



a. Should arc line selection require precise picking with a mouse?



b. Should "grab handle" be used to aid arc selection?



c. How should vertex movement be designed?

Figure 2. A prototype was used to answer design questions about control flow arcs in the Graphical Programming Language Tool

The GPL prototype also helped provide insight into some questions about interface modality. We wondered whether modes were necessary in the design, and we needed information about how often modes would be used and whether they would cause user confusion. The GPL tool is used to create supervised flow diagrams by creating various kinds of node symbols (for states) on the screen and then connecting them with control flow arcs. Considerable detailed development must then be done for both nodes and arcs. Users of the prototype showed us that there is enough cognitive difference between development of nodes and development of arcs to justify separate development modes for these two kinds of objects. Further questions of modality arise when one considers whether the user should be able to select a node type once and create several instances on the screen or whether the user must select the icon for a diagram symbol each time a symbol instance is added to the screen. Also, since further design information must be given for each symbol, can it be given successively for a number of existing symbols or must the information be given as a symbol is created?

Exercise of the prototype by potential DMS users showed that both modal and non-modal usage was desirable, depending on the user and the situation. The prototype was subsequently used to make the design as flexible as possible for further exploration of both approaches.

Because the GPL tool is fairly large and complex, only a small portion of the functionality of the prototype is shown here. Figure 3 shows the GPL tool prototype after construction of a supervised flow diagram consisting of a start symbol (small circle), a dialogue transaction (large circle) called GetCommand, a dialogue-computation function (box with inscribed circle) called DoCommand, and a triangular return symbol. The various symbols in this diagram were created by selecting the corresponding icon at the bottom of the screen and positioning a new instance of the desired symbol with the mouse. The arcs were drawn by selecting the arrow icon (the horizontal arrow just to the right of the triangle in the lower right part of the screen), which places the prototype in a different mode (the arc development node), and choosing the beginning and ending nodes for each arc.

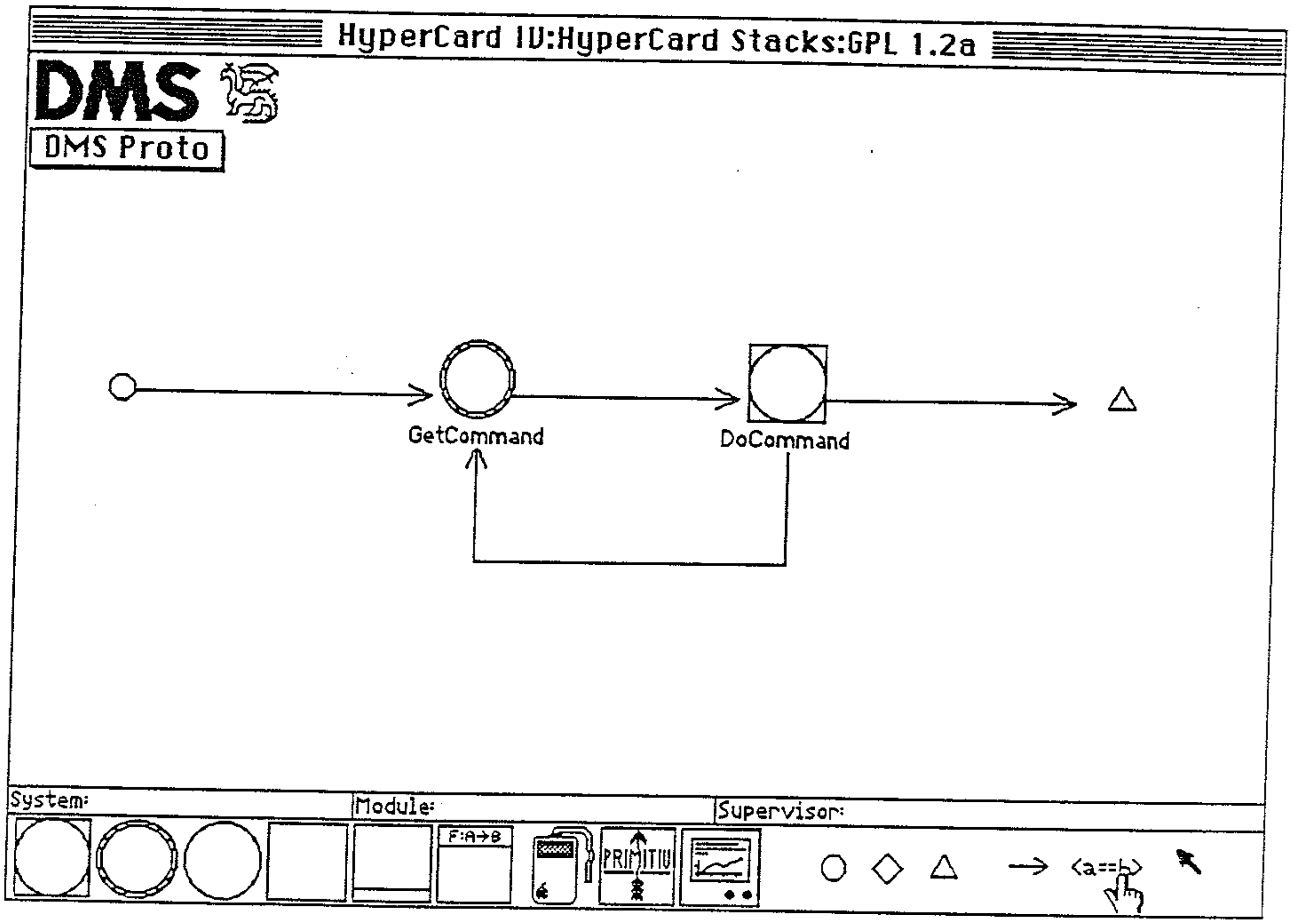


Figure 3. HyperCard GPL prototype screen

Figure 4 shows the user selecting a symbol to develop further. This is possible when the prototype is in symbol deveopment mode, as opposed to arc development mode. Because each symbol on the screen is associated with a HyperCard button, mouse selection capability was already provided.

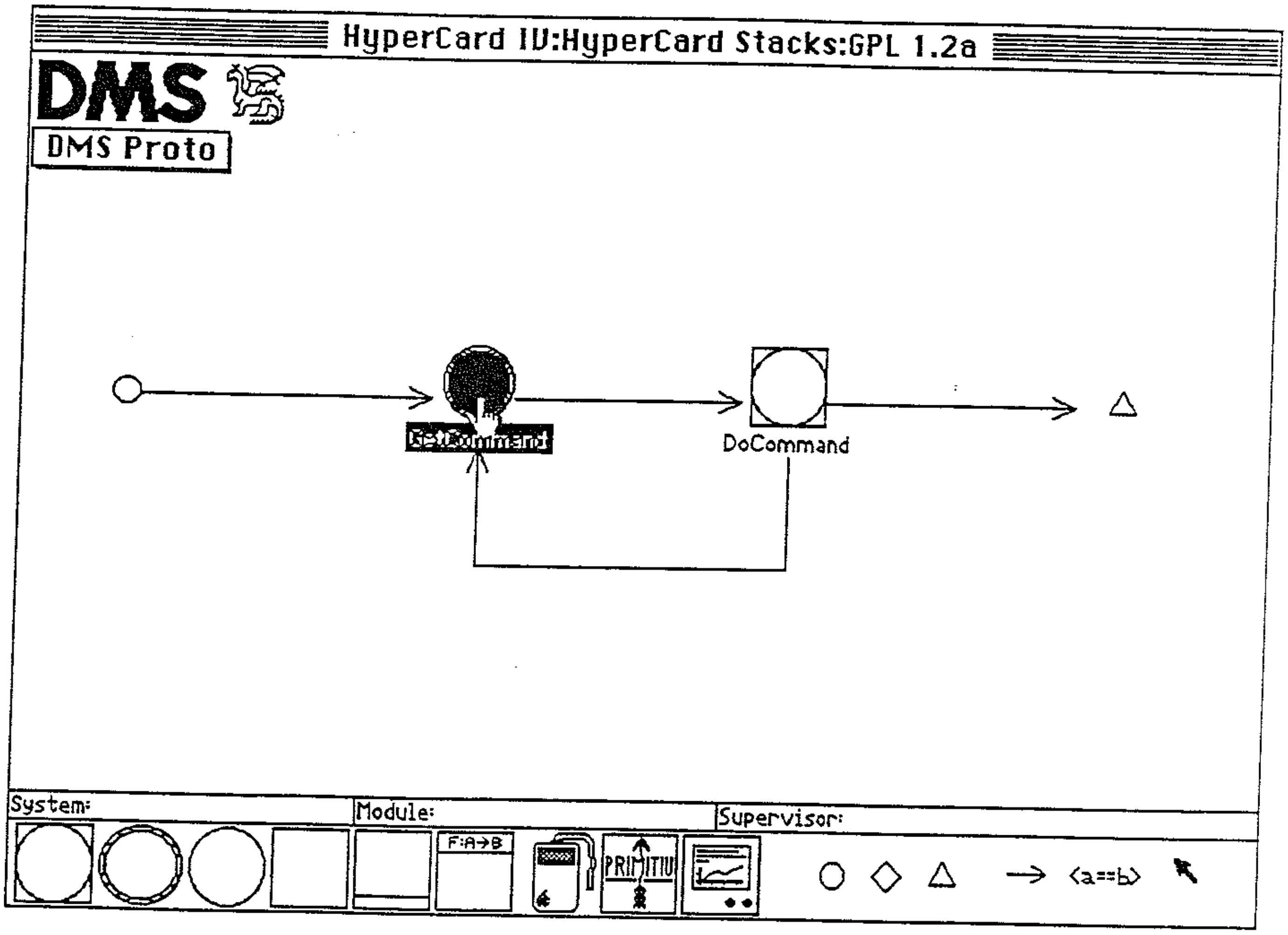


Figure 4. Selecting a symbol to develop

By employing an X-function, a fast pop-up menu was added to the prototype. In Figure 5, the user is selecting the command "Symbol Info..." which allows the user to assign names, as well as documentation and other important information, to symbols.

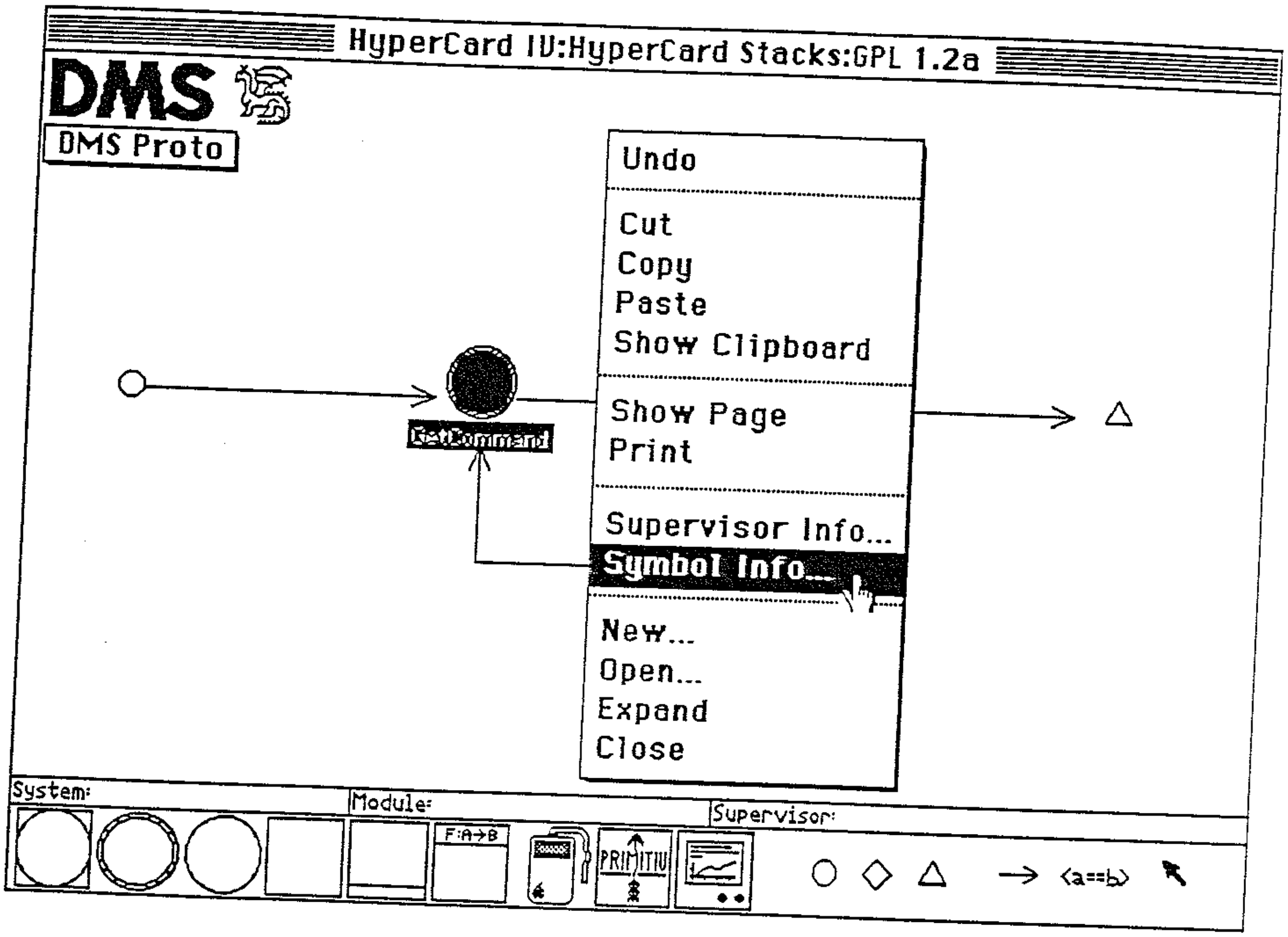
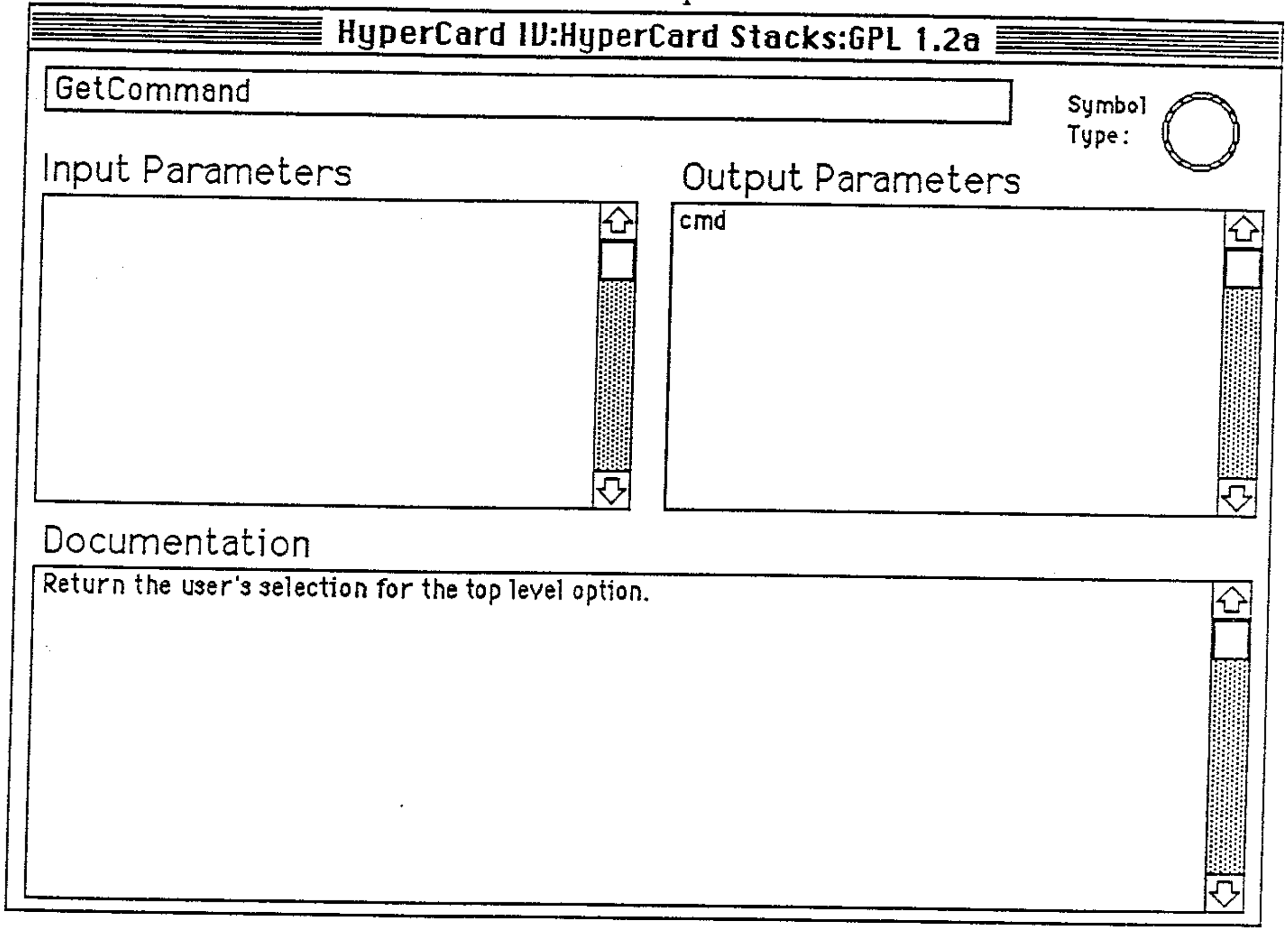


Figure 5. Selecting "Symbol Info..." from pop-up menu

After selection of the "Symbol Info..." command, the prototype moves on to another card, as shown in Figure 6, which is a prototype of the symbol information box. The user may then enter the data to be associated with the selected symbol and return to the GPL prototype screen.

Although there is not space to show them here, most of the commands on the pop-up menu and all of the iconic commands were available in the prototype. Having such a complete prototype as a guide proved invaluable during the construction of the actual product.



GetCommand

cmd

Return the user's selection for the top level option.

Figure 6. Prototype of symbol information box

A more general use of the GPL prototype was to experiment with the screen layout for the tool: the location of the command-icon bar, the visual design of each icon, the relative location of textual labels for user-created GPL objects, the location of information such as the names of the user interface objects and screens being created and names of their related software modules. The prototype allowed the interface developer to view these screens and objects, to evaluate the general use of screen space, and to gain some "user" experience. It is interesting to compare the final GPL prototype screen (Figure 3) with the same screen as implemented later in Smalltalk (Figure 7). This comparison reveals a side benefit of our prototyping: the porting of icons and other graphics in the prototype, via MacPaint files on the clipboard, directly into Smalltalk "forms." This gave us some feeling for the advantage of evolutionary prototyping; the detailed icon design work was saved and few changes were needed during the crossover.

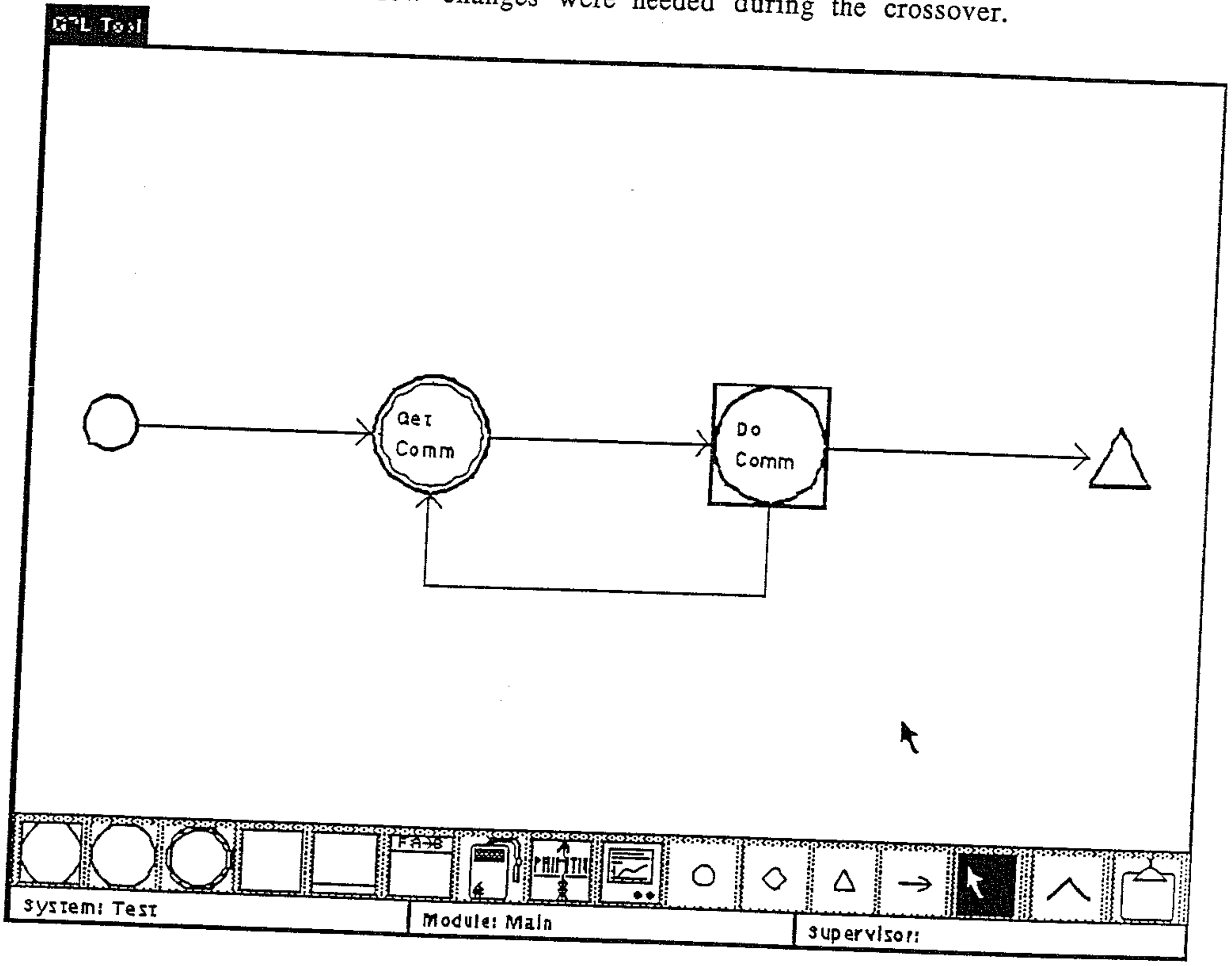


Figure 7. GPL tool screen as implemented in Smalltalk (compare with HyperCard prototype screen, Figure 3)

5.3 Local Prototyping

During the design process, it was often easier and faster for the developer and evaluator to use a "quick and dirty" mockup of a small part of a tool to answer isolated interface design questions than to wait for a full prototype. While global prototypes are rather complete, our local prototypes focused on a few specific questions and were not intended to be extendible.

The iconic menu tool, an example of a tool that benefitted from local prototyping, uses graphics and text editors to allow the interface developer to create objects that represent a one-from-many selection menu and its choices. For each choice the interface developer must use the "Develop Input" command to specify a token value to be sent to the rest of the system when an user selects that icon with the mouse. Developers of the tool were faced with the mode-related question of whether to develop input for choice icons one at a time or to do this for groupings of icons.

In the iconic menu tool we wished to offer the interface developer an immediate and direct ability to test a menu, through a "Test" command, as it is being developed. Use of the prototype revealed another similar mode question. Should the "Test" command allow testing of several successive user inputs or should each test input take the tool back to being ready to develop the menu further? It was important to be able to answer these questions on the basis of some user experience. For the sake of consistency we would be committed to using the results whenever they applied throughout all our tools.

A brief description of use of the iconic menu tool prototype is given in the following paragraphs. All prototyping tools have at least two modes of operation -- a "develop" mode in which interface designs are developed and a "test" mode in which people assuming the end-user role test these designs. When the example prototype is started, the screen in Figure 8 appears, showing the tool in Develop Mode with a prompt containing a single menu choice icon (in the lower left part of the screen). **[Note to editor: Most figures in this sequence and some earlier figures can be somewhat reduced in size.]**

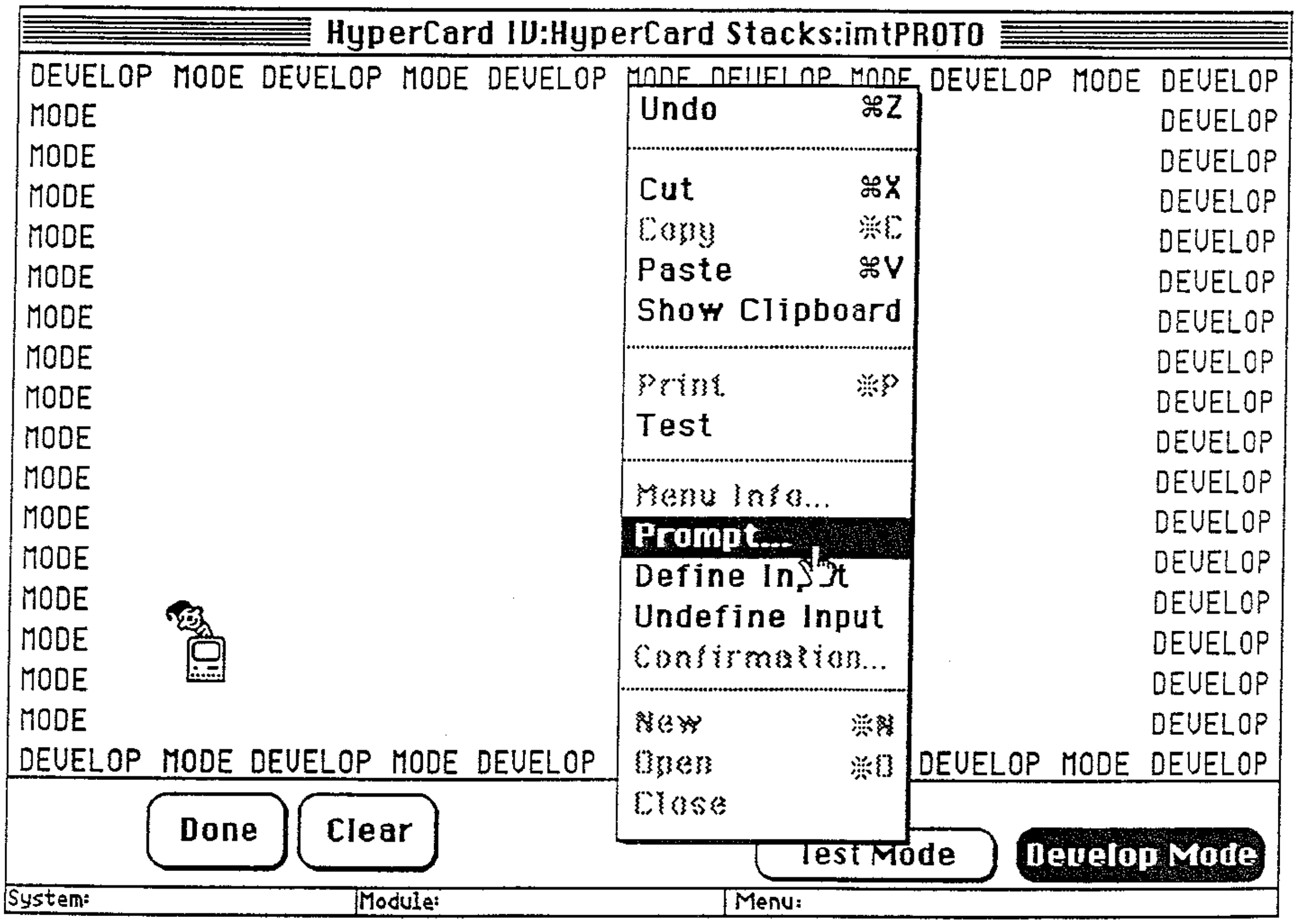


Figure 8. Iconic menu tool local prototyper screen

The HyperCard button tool is used to add several new choice icons to the menu, as shown in Figure 9. At this point these icons are only objects within a display and are not sensitized as menu choices selectable by the end-user.

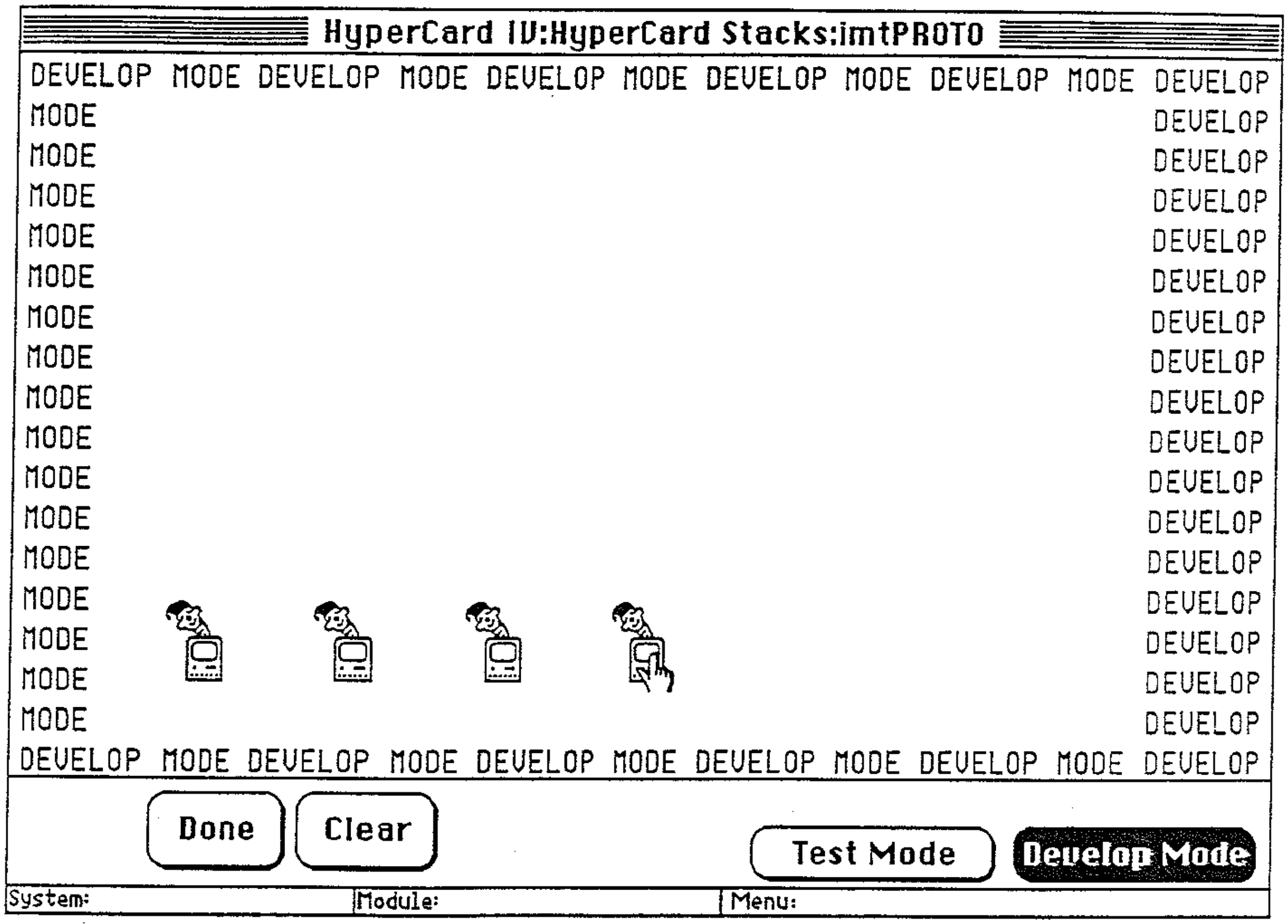

HyperCard IV:HyperCard Stacks:imtPROTO
DEVELOP MODE DEVELOP MODE DEVELOP MODE DEVELOP MODE DEVELOP MODE DEVELOP
Done
Clear
Test Mode
Develop Mode
System:
Module:
Menu:

Figure 9. Several choice icons in menu prompt

To define these icons as possible end-user inputs the interface developer selects the three new icons by using a shift-select operation similar to that used by the Macintosh Finder. Figure 10 shows the icons selected as a group.

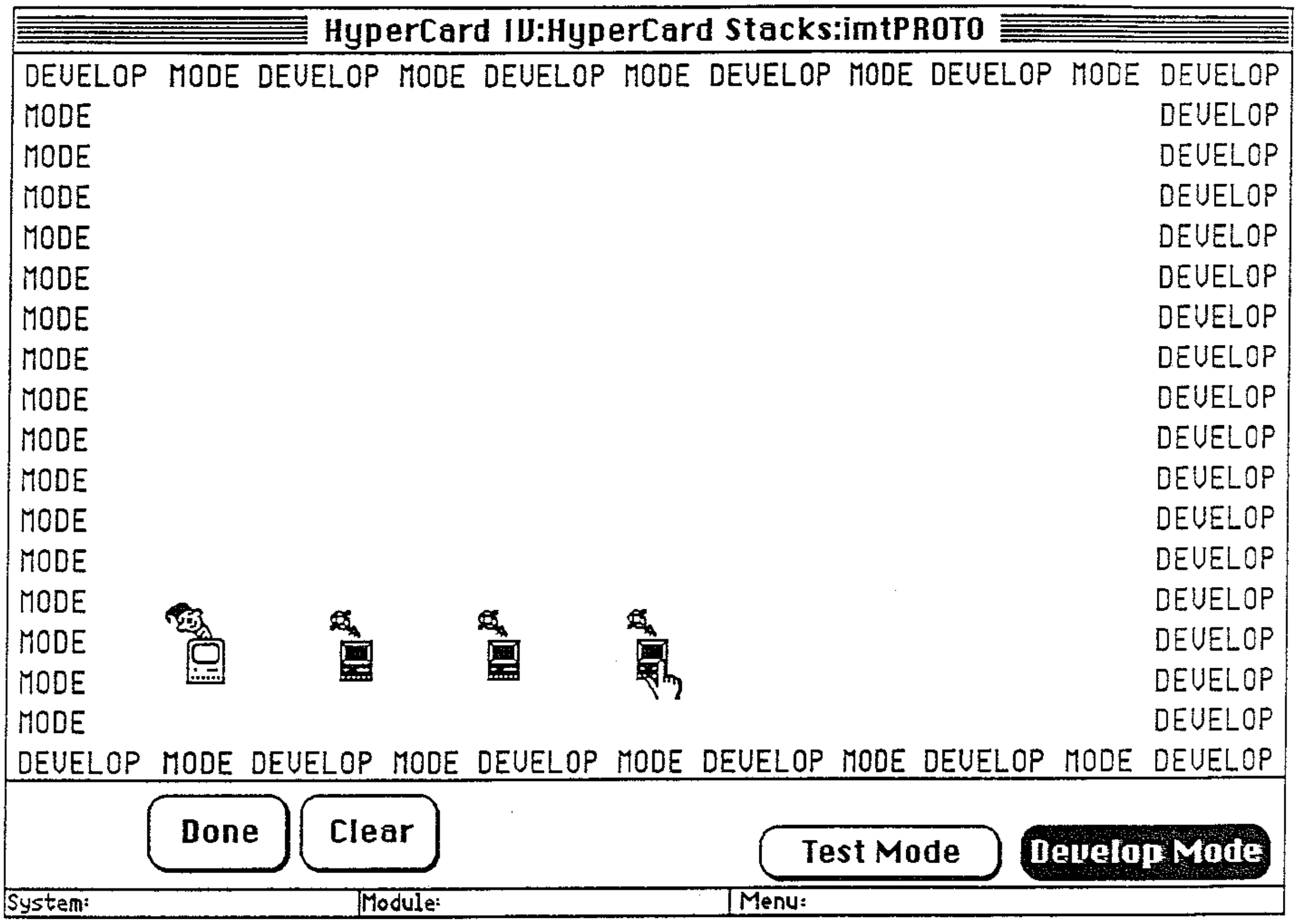
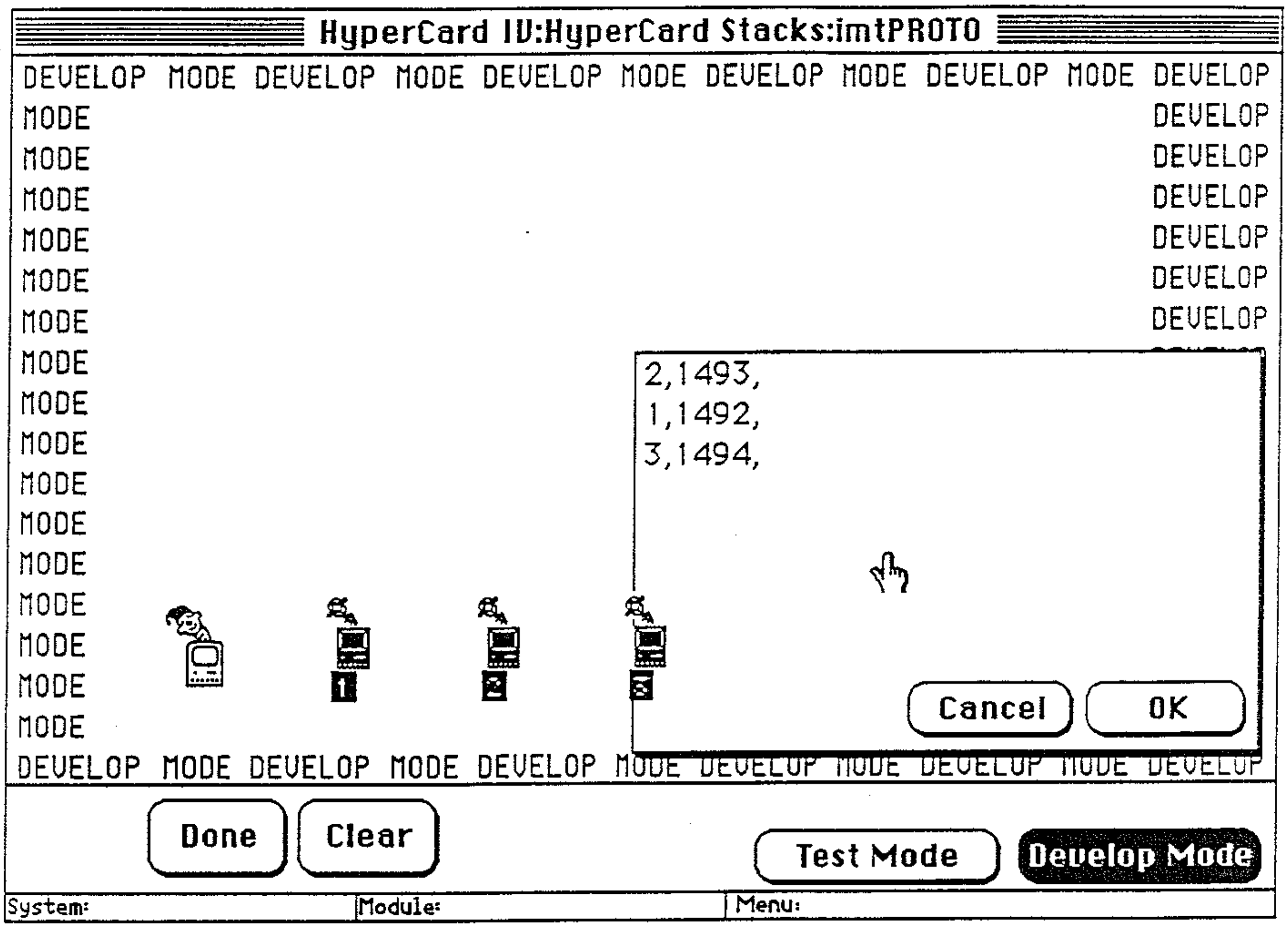

HyperCard IV:HyperCard Stacks:imtPROTO
DEVELOP MODE DEVELOP MODE DEVELOP MODE DEVELOP MODE DEVELOP MODE DEVELOP
Done
Clear
Test Mode
Develop Mode
System:
Module:
Menu:

Figure 10. Group selection of choice icons by developer

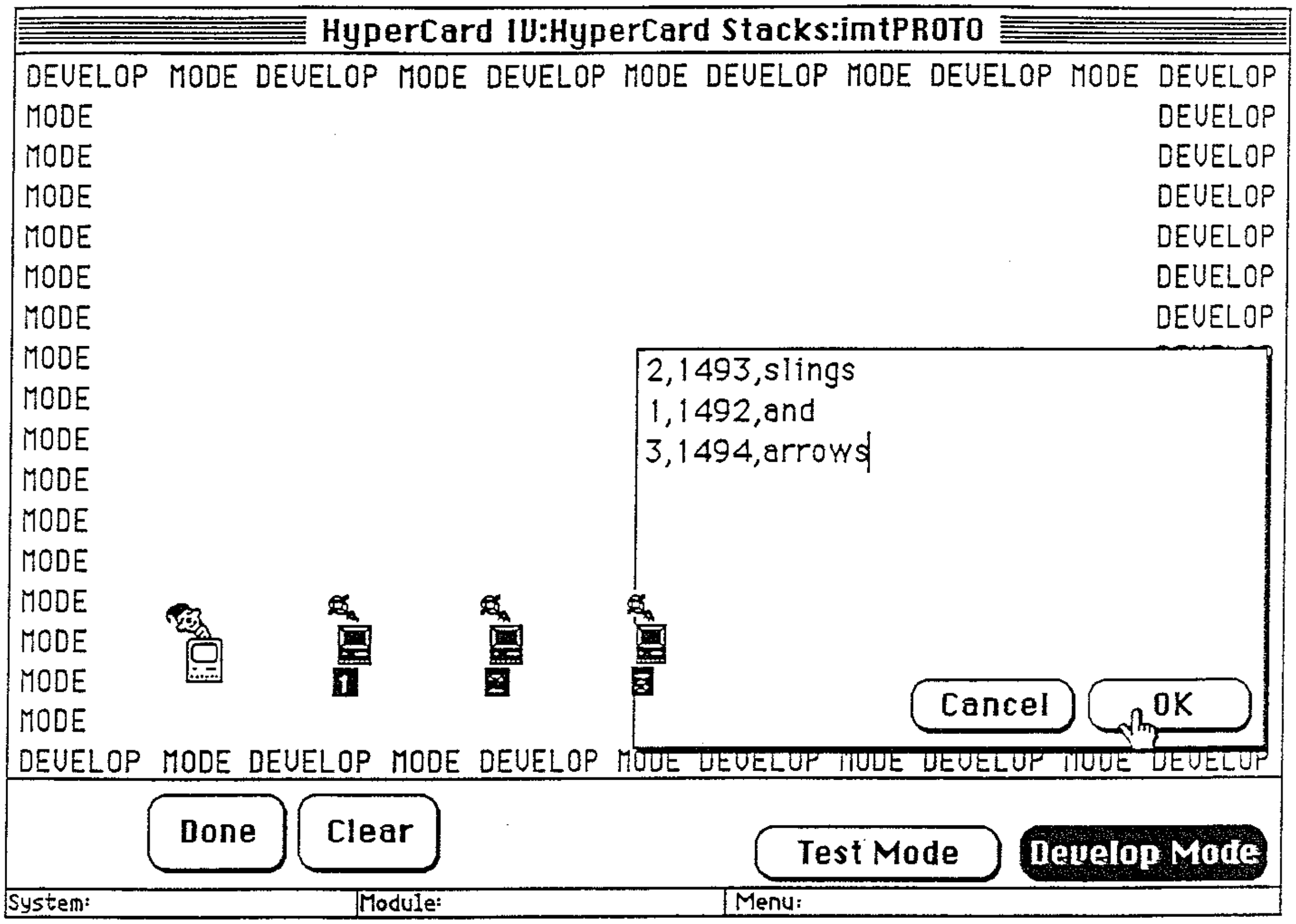
A pop-up menu command, "Develop Input," is then issued by the interface developer causing the screen shown in Figure 11 to appear. In this screen, a box has been brought up containing a label visually showing the correspondence between each of the three icons selected by the developer and an internally used HyperCard button identifier. The labels, used only during the development of the menu, have defaulted to "1", "2", and "3".



```
2,1493,
1,1492,
3,1494,
```

Figure 11. Dialogue box showing choice icon labels and internal HyperCard button identifiers

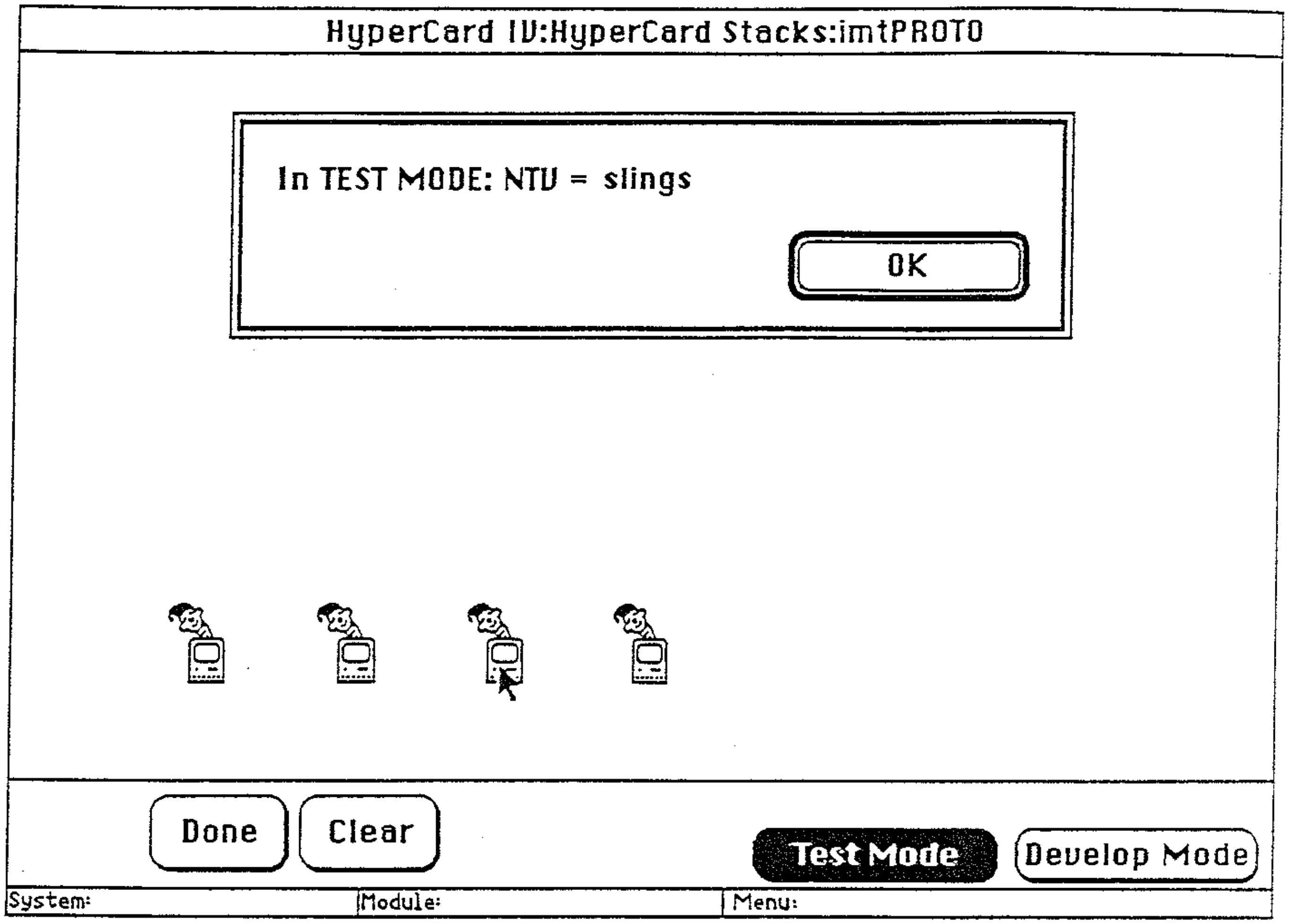
The developer may assign more appropriate names. The four-digit button identifiers are presented only to the developer. They do not appear in the final tool implementation. As these icons are new, the field after the second comma in each line is blank. This is where the developer enters (and edits) the token value to be returned upon selection of the icon at run-time for each of the three selected icons, as shown in Figure 12. A click on the "OK" button saves the changes and gets rid of the box.



2,1493,slings
1,1492,and
3,1494,arrows

Figure 12. Token values specified for end-user selectable icons

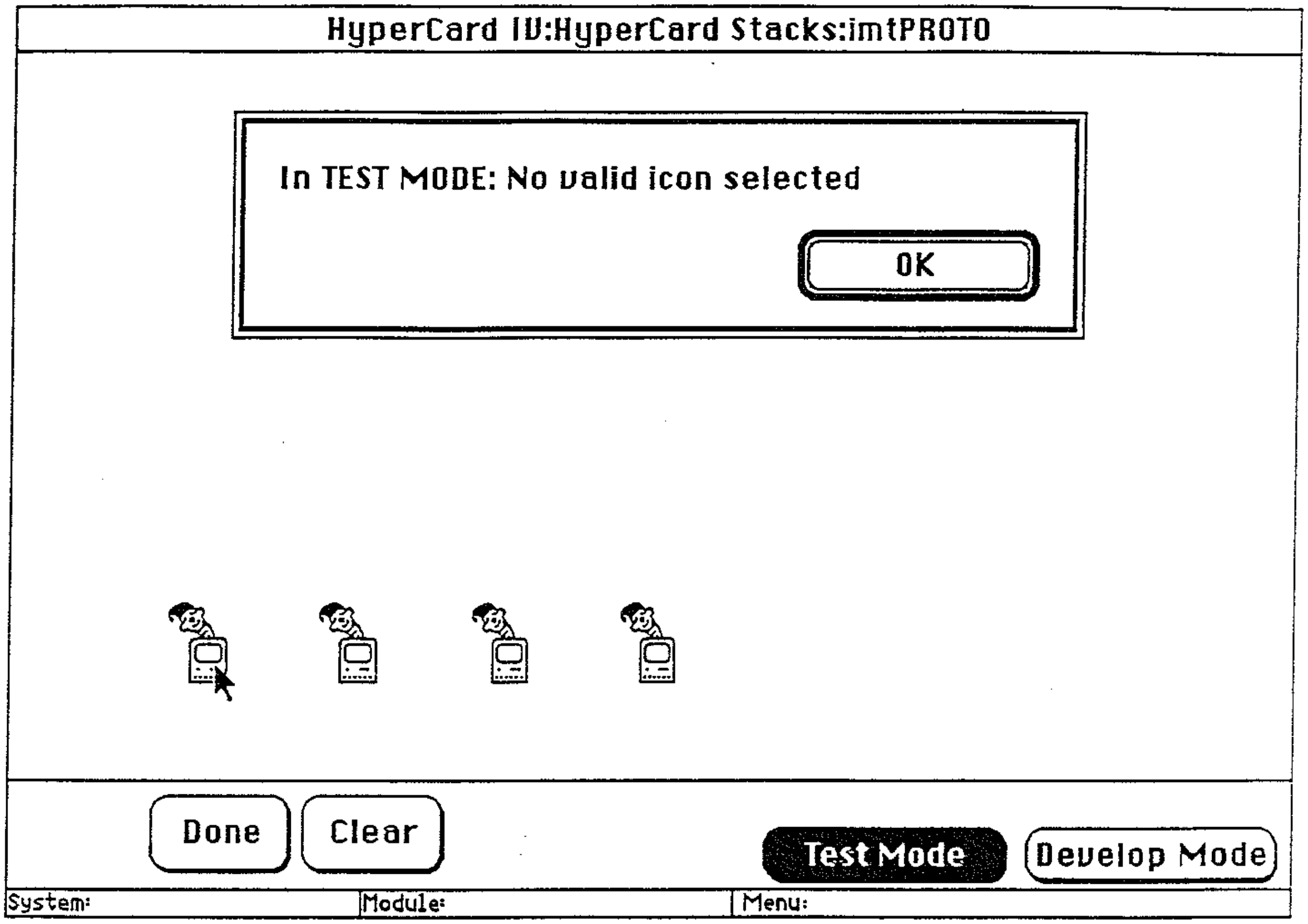
By putting the tool in Test Mode, the developer can immediately test the menu. The bordering of screen has changed to make the developer aware of the mode change. By clicking on an icon or screen area as an end-user would, the developer can see what token value would be returned to the rest of the software. Figure 13 shows the successful pick of icon "2" and the normalized token value (NTV) for that icon, the character string "slings."


HyperCard IV:HyperCard Stacks:imtPROTO

In TEST MODE: NTV = slings

Figure 13. Test mode result showing normalized token value (NTV) for successful pick of icon "2"

Figure 14 shows the results of selecting the first icon, for which a normalized token value has not been defined.


HyperCard IU:HyperCard Stacks:imtPROTO

In TEST MODE: No valid icon selected

Figure 14. Test mode result for pick of first icon, for which a normalized token value is not defined

The token value definition for a given icon or group of icons may be deleted by selecting the icon or icons and choosing the pop-up command "Undefine Input." Figure 15 shows the results of applying this command to icon "2". Figure 16 shows that in Test Mode, that icon is no longer considered a valid choice.

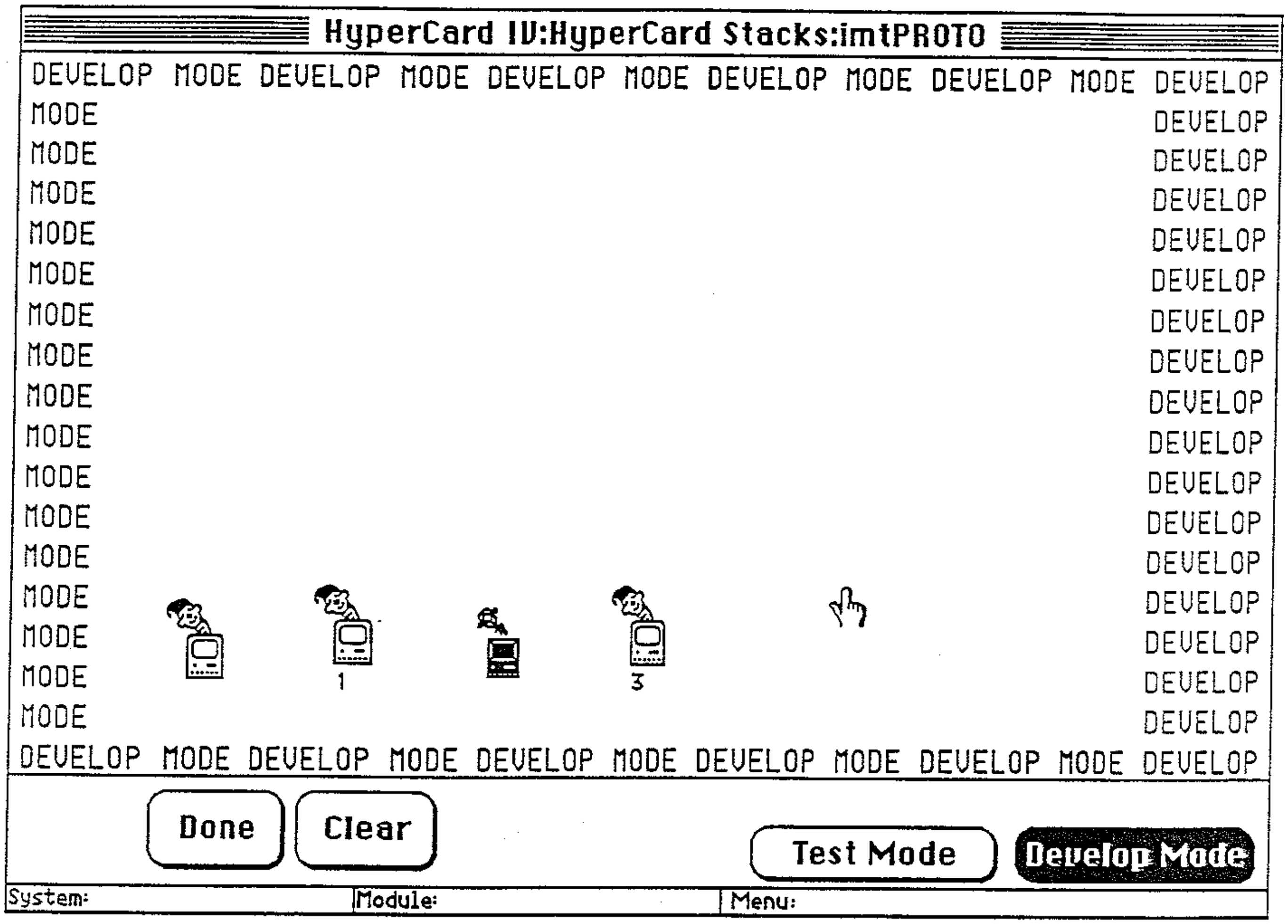
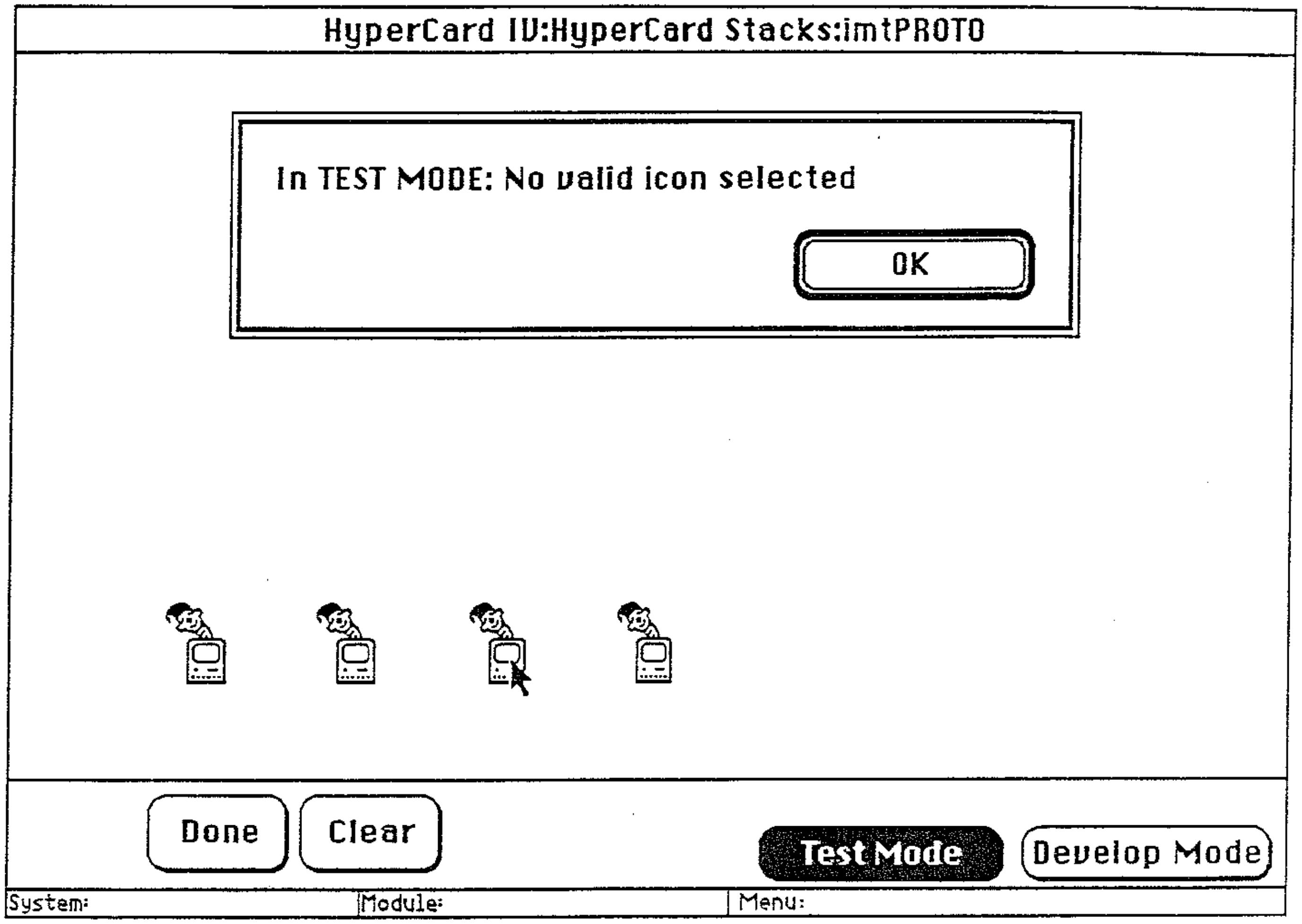

HyperCard IV:HyperCard Stacks:imtPROTO
DEVELOP MODE DEVELOP MODE DEVELOP MODE DEVELOP MODE DEVELOP MODE DEVELOP
1
3
Done
Clear
Test Mode
Develop Mode
System:
Module:
Menu:

Figure 15. The result of selecting and applying "Undefine Input" to icon "2"


HyperCard IU:HyperCard Stacks:imtPROTO

In TEST MODE: No valid icon selected

Figure 16. Test mode result for subsequent pick of icon "2"

5.4 Lessons Learned

We encountered two significant questions in using prototyping as a part of the development process:

- How much of the design should be included in the prototype?
- How far should development of the prototype be carried?

The answers are different depending on whether the prototype is global or local.

Before using the global prototype of GPL there was an attitude toward the first question that the prototype, being only a "first draft," needs only a loose resemblance to the real design. However, we found that to have an effective global prototype--one that can be used by evaluators and users to get a reasonable "feel" for the evolving system--it is necessary to include most of the functions that appear in the design of the system. In addition, the prototype should be as similar in appearance as possible to the real system design. Differences between the two led to misunderstandings by the evaluator and potential users. They interpreted the prototype as a literal instantiation of the design, and differences produced surprises and some disappointment. Some functions, however, were so difficult to produce using HyperCard that we chose not to expend the effort to include them in the prototype. Instead, we opted for a careful pencil-and-paper prototype of those functions. We encountered several examples of this: one was the "undo" function in GPL. Programming "undo" in HyperCard proved to be very difficult, and did not seem to be critical to the overall "feel" of the prototype. Other examples were the "pan" function (to move through diagrams too large to fit on one screen), the movement of vertex points in segmented arcs, and use of multiple windows. In these cases the limitations of HyperCard prevented the prototype from being as complete and accurate as we desired. Pop-up menus might also have been omitted but they were central to the design. They turned out to be difficult to prototype and the HyperCard code was not aesthetic.

We found it relatively easy to answer the second question, by following this criterion: cease development of the prototype when the effort of prototyping exceeds that of implementation of the actual system. Early in the development process, we found the prototype to be invaluable in helping us make design decisions, but as the design progressed, we began to find that conditions shifted so that using Smalltalk for the implementation started to be faster and easier. The shift in conditions was due to the trade-off between learnability and expressive power of the two languages. A developer was very proficient with HyperCard in two weeks. Smalltalk has a much steeper learning curve; the difference put the GPL prototype about two months ahead of implementation by the end of the third month. Beyond that there was a "crossover" and the greater expressive power and flexibility of Smalltalk made it easier to update the interface design directly in the implementation, rather than the prototype.

In our project the learnability factor was important because we learned both HyperCard and Smalltalk as we proceeded. Even if both had been known at the outset, global prototyping would have had a role but the crossover described above would have occurred somewhat earlier. It would always be easier and faster to do some screen mockup with HyperCard, and local prototyping continues to be useful beyond the crossover point.

Our conclusion is that global prototyping served a planned role in our development process very well. It indeed allowed observation of early behavior and subsequent adjustment of the design. The prototype allowed interface developers to dream a little and encouraged them to play with design alternatives. Numerous instances of design incompleteness, ambiguity, and inconsistency were revealed, allowing us to resolve them with the developers while there was still enough flexibility to make changes.

In our example, the choice of languages aided the transition to implementation. HyperCard has a strong object-oriented flavor; an input event, such as a mouse pick of an object, activates a programmed action. We deliberately exploited this in trying to make the prototype code as object-oriented as possible in order to match the object orientation of Smalltalk. For example, each card button corresponds to an object that contains the knowledge of how to display its own instances.

Our conclusions about local prototyping were also positive. The iconic menu tool prototype was constructed in about six hours and it gave us some very good information for decision making. In sticking to the predefined purpose of this prototype, we had to resist making it more complete. It would have been wasteful to develop the local prototypes beyond what was needed to answer specific design questions.

In sum, our prototyping experience was strongly supportive of our development process. We were affected, however, by the limitations of HyperCard. We wished for a more powerful prototyping tool and one more integrated into the development environment, to make the transition from prototype to implementation more evolutionary.

6. SYSTEM DEVELOPMENT ISSUES

6.1 Methodology and Life Cycle

Rapidness of prototyping. Gould and Lewis [1985] present these principles for interface development:

- early and constant focus on users and tasks,
- empirical evaluation of interface usability, and
- iterative refinement throughout the life cycle.

However, as pointed out in section 1.1 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 an intermittently executable prototype 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 section 3.4 on prototype executability.) Effective development, especially if interface quality is an important factor, requires a process that allows a much finer resolution for iteration. 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 paper (e.g., Bally, Brittan, and Wagner [1977]).

A Star Life Cycle. In seeking a suitable development methodology, no place is found for integrating rapid prototyping concepts into traditional top-down, stepwise decomposition methodological models [Mantei 1986; Mantei and Teorey 1988]. In response to this need, Hartson and Hix [1989b] have proposed a new life cycle to accommodate rapid prototyping as an explicit development activity. Named the "star" life cycle because development activities are clustered in a star configuration about *evaluation* as a central activity, it is a departure from the traditional approach in several ways. It is a process of real cycling and less a linear process. Developers can enter the cycle almost anywhere. 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 section 1.2 on prototyping as 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. A prototype is specific about *how* interface features are designed. From this initial design many of 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. The star life cycle, as a development cycle for user interfaces, often exists within a larger, and possibly different, life cycle for overall target system development.

Alternating modes of development activity. In case studies done in the Dialogue Management Project [Hartson and Hix 1989b], we observed that, overlaid upon local cycling and phases of the life cycle, there is an interesting progression by developers through various modes of development activity.. Throughout the overall interface development process there appears to be a series of alternating waves of upward (bottom-up) and downward (top-down) progressions through the levels of abstraction in the design. We observed that less experienced as well as more experienced developer subjects, when not constrained by a particular methodology, often start 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 the developmental view of psychology. Some successful user testing is often accomplished even with these very early tentative scenarios. Many developer subjects then produce a state diagram or similar flow-chart-like representation to show a more general view of sequencing among interface screens. This is a top-down structuring activity. Perhaps some attention is given next to details in the design of the screen objects at a very low level of abstraction. At some point, we observed developers returning to a top-down mode, working 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 user testing of details at a low level of abstraction entered in, using a prototype interface. Similar observations about natural interactive system development were made independently by Carey [1988].

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, concrete, and behavioral; they tend to reflect the user's view. Downward activities are more theoretical, analytical, abstract, and structural; they tend to reflect the system developer'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 prototype screen design. A control structure to support prototype sequencing is developed top-down, returning to bottom-up user testing of details, followed by top-down modeling and abstraction to restructure the design, and so on.

Management Concerns. Management of software development has traditionally been based on a linear progression through phases of the process, each phase punctuated with a milestone. Upon reaching a milestone, management can "sign off" its approval and all parties concerned have tangible evidence of progress.

A truly iterative cycle such as the star life cycle, having no apparent end, can rightfully cause serious management concerns about how to control the process. There is no fixed order for development activities. Isn't this a formula for chaos? No milestones mark the end of development phases. How does one know when the process is completed? How can one be sure that iteration will result in convergence on a good design?

Answers to such questions are now emerging from a subdiscipline called usability engineering [Whiteside, Bennett, and Holtzblat 1988]. Management can sign off on a different kind of goal. Usability requirements, specific measurable criteria for user performance and satisfaction, are stated at the outset. Management can approve parts of the design as testing in the evaluation activity shows corresponding usability requirements to be met.

Convergence on improved designs is served by techniques such as impact analysis [Good, Spine, Whiteside, and George 1986], which involves measuring time spent by user subjects with interface problems (e.g., time to recover from an error). Such data direct attention to those parts of the interface that detracted from task performance. Solutions to those interface problems have the highest potential for helping to meet usability performance requirements.

Integration. Because the influence of the behavioral scientist is finally, and rightfully, becoming a factor in development of interactive systems, a final cautionary note about development methodology is warranted. Some methods 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. An interactive system is not just an interface. There is a great amount of non-interface software with which the interface must be integrated in an interactive system; methodologies without connections to this software and methods for its development cannot be considered as serious possibilities to meet real world development needs.

6.2 Tools and User Interface Management Systems

Although prototyping can be accomplished without the aid of automated support, management of the developing design can quickly become intractable. Use of computer aids in constructing and documenting designs and prototypes can help a great deal if these tools are well designed. These tools can be used to maintain, for example, information about configurations, various versions, and reasons for design changes. This information, gathered as tools are used (both by developers and by 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. User Interface Management Systems (UIMS) are becoming a key tool in this capacity. Rapid prototyping is now an integral part of most UIMS [Lewis, Handloser, Bose, and Yang 1989]. Modern UIMS, some of which were described briefly in section 4 on example systems and approaches, allow many alternative designs to be tried in a short period of time (hours as opposed to days or weeks). Although many UIMS concentrate on particular interaction styles or are limited to certain classes 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. Many include simulators, or interface definition interpreters, which allow the interface to be designed and prototyped entirely within the UIMS.

Since many UIMS 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 developer to view or execute the interface as it is being designed. Because of the ease with which they permit 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.

6.3 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 this ability? The essence of the answer is that they will evaluate designs early in the development process. There are many ways that this can be done; some are described in what follows. A thorough coverage of user-based testing for the evaluation of system and interface designs is well beyond the present scope. 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, both in general, and specifically in relationship to rapid prototyping, which emphasizes case studies, verbal protocol thinking, and critical incident analysis (described briefly below).

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 measures of 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 interpreted within specific design situations. The process of testing individual interface points and features, however, is not the effective evaluation process needed 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 there is 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 user to fit the system! The 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 for one 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 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. 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 subsection on traditional controlled experimentation under section 6.3 on design evaluation, 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 development cycle. Experimental sessions should be videotaped so they can be replayed as needed for analysis. One important technique for extracting information from the 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 identifying significant data from the noise, 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, Herda, and Itzfeldt 1978].

Post-session interviews and *questionnaires* can be used in case studies as 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 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 user feedback, too. Each screen of the interface can be augmented with an additional 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 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 details that may be forgotten before the post-session interview. It allows the user to express feelings at the moment they are experienced. Even a short delay can diffuse or defocus those feelings. Some problems are a problem to a user for only a short time. After that, the marvelously flexible human may adapt and smooth over the rough spots that 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 to build automatic instrumentation into the prototype for monitoring 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 user's task.

7. TECHNICAL PROBLEMS AND SOLUTIONS

7.1 Building Tools to Prototype 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 developed interfaces (and systems) is of great value in their 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. Because 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, even a stubbed system must still be syntactically complete and correct to compile and run, or even to be interpreted. Even though getting syntax correct is not a difficult problem, it involves attention to detail that has nothing to do with usability and much of the detail will be changed later, anyway. This is a major drawback with interface prototypes that are programmed, either in a programming language or a high level dialogue language.

Fortunately, many new prototyping tools have high level, non-programming, direct manipulation interfaces for developers. However, these tools must produce code or data that directs the course of execution in "test mode". If the high level specification of the interface is incomplete, the code produced may be correspondingly so. The problem now properly belongs to those who design and construct the prototyping tool.

In sum, 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 limbs and organs can be molded into a single correct and complete design. Allowing incomplete designs to be executed effectively as prototypes is one of the difficult technical problems in providing a usable rapid prototyping facility. It is a serious challenge to the computer science role and may not be appreciated as such by the behavioral scientist.

7.2 The Information Management Problem

During design and prototyping phases of system development, an enormous amount of information is produced. Because the rapid prototyping approach introduces many design variations in an environment in which more than one development phase can be active concurrently, the problem of keeping track of information is multiplied. The problem of information loss also becomes an important consideration 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, and Green 1983]. The reasoning behind a given design or design change and the history of revisions are often not available to implementers 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, in particular during design and prototyping phases, involve two major components. One component is use of *automated tools* (e.g., UIMS) interactively to 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 1986] 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 each maintain the representation which best suits its needs.

7.3 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, developers 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 developer 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 incrementally constructing whole-system, continuously executable prototypes. 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 interface and system objects. Object-oriented programming [Cox 1986] is being increasingly used for developing the 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].

8. CONCLUSIONS AND FUTURE

Rapid prototyping is a relatively new concept in the software industry. Yet, as it increases in popularity and maturity, it is already changing the way interactive systems are developed, speeding up the process and making a better and more usable product. Increasingly powerful automated tools are becoming available. These tools 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 can be produced faster and less expensively and that are more satisfactory to 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 user roles for human-computer interface design and evaluation, roles lacking in current methodologies. They will explicitly include iterative refinement in the life cycle.

In sum, improved tools set in better software development environments, together with rapid prototyping methodologies and a body of practical experience will combine to revolutionize interactive system development, and especially development of the human-computer interface.

ACKNOWLEDGEMENTS

The authors wish to thank Pat Cooper and Jo-Anne Lee Bogner for their cheerful and enthusiastic care and feeding of this manuscript over the long period of its growth and maturation. Special thanks for help of many kinds also go to Deborah Hix and Antonio Siochi. We appreciate the additional help of James Callan, Beverly Williges, Roger Ehrich, Roger Chen, and Jeff Brandenburg. 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; from the Virginia Center for Innovative Technology; and the Software Productivity Consortium, our sponsor client for the DMS 3.0 as described in section 5 on examples from a development project.

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 Systems. *IEEE Transactions on Software Engineering, SE-4*, 4, 270-276.

Ehn, Pelle. (1989) *Human Centered Design and Computer Artifacts*, forthcoming.

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.

Good, M., Spine, T., Whiteside, J., and George, P. (1986) User-Derived Impact Analysis as a Tool for Usability Engineering. *Proceedings of CHI '86* , Boston (April), New York: ACM, 241-246.

Gould, John D., and Lewis, Clayton (1985) Designing for Usability: Key Principles and What Designers Think. *Comm. of the ACM, 28*, 3 (March), 300-311.

Gray, Philip D.; Kilgour, Alistair C., & Wood, Catherine (1988) Dynamic Reconfigurability for Fast Prototyping of User Interfaces. *Software Engineering Journal*, (November), 257-262.

Green, Mark (1985) The University of Alberta User Interface Management System. *Proceedings of SIGGRAPH '85, 12th Annual Conference* San Francisco, CA: ACM, 205-213.

Gregory, S. T. (1984) On Prototype vs. Mockups. *ACM SIGSOFT Software Engineering Notes, 9*, 5, 13.

Gutierrez, Oscar (1989) Prototyping Techniques for Different Problem Contexts. *Proceedings of the CHI '89* , Austin, TX (April), New York: ACM, 259-264.

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. (1980a) A Prototyping and Simulation Approach to Interactive Computer System Design. *Proceedings of the Design Automation Conference* Minneapolis, MN.: ACM, 572-578.

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

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

Hartson, H. Rex, and Hix, Deborah. (1989a) Human-Computer Interface Development: Concepts and Systems for its Management. *ACM Computing Surveys, 21*, 1 (March), 5-92.

Hartson, H. Rex, and Hix, Deborah. (1989b) Toward Empirically Derived Methodologies and Tools for Human-Computer Interface Development. To appear in *International Journal of Man-Machine Studies.* [Note to ed: Might have more specific reference by the time present paper goes into production.]

Hartson, H. Rex, Johnson (Hix), Deborah, and Ehrich, Roger W. (1984) A Human_Computer Dialogue Management System. Proceedings Human-Computer Interaction--Interact '84, B. Shackel (ed,), London (September) Elsevier Science Publishers B. V. (North Holland), 379-383.

Hill, Ralph D. (1987) Event-Response Systems: A Technique for Specifying Multi-Threaded Dialogues. *Proceedings of the ACM CHI + GI '87 Conference* Toronto, Ontario, Canada (April), New York: ACM, 241-248.

Jacob, Robert J. K. (1982) Using Formal Specifications in the Design of a Human-Computer Interface. In *Proceedings Human Factors in Computer Systems* Gaithersburg, MD.: ACM, 315-321.

Jacob, Robert J. K. (1985) An Executable Specification Technique for Describing Human-Computer Interaction Chapter 8 of H. Rex Hartson (Ed.), *Advances in Human-Computer Interaction, Vol. 1* Norwood, NJ: Ablex, 211-242

Jacob, Robert J. K. (1986) A Specification Language for Direct-Manipulation Interfaces. *ACM Transactions on Graphics* (October), 283-317.

Lewis, T. G., Handloser, Fred, III, Bose, Sharada, and Yang, Sherry. (1989) Prototypes from Standard User Interface Management Systems. *IEEE Computer, 22*, 5 (May), 51-60.

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

Mantei, Marilyn M.., and Teorey, Toby J. (1988) Cost/Benefit for Incorporating Human Factors in the Software Lifecycle. *Comm. of the ACM, 31*, 4 (April), 428-439.

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

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.

Myers, Brad A. (1988) *Creating User Interfaces by Demonstration (Perspectives in Computing, Vol. 22)*, Boston:.Academic Press, Inc.

Myers, Brad A. (1989) User Interface Tools: Introduction and Survey. *IEEE Software*, 6, 1 (January), 15-23.

Nielsen, Jakob (1987) Using Scenarios to Develop User Friendly Videotex Systems. *Proceedings NordDATA87 Joint Scandanavian Computer Conference* (June).

Olsen, D. R., Jr. (1986) Mike: The Menu Interaction Kontrol Environment. *ACM Transactions on Graphics* (April), 318-344.

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

Richards, John T., Boies, Stephen J., and Gould, John D. (1986) Rapid Prototyping and System Development: Examination of an Interface Toolkit for Voice and Telephony Applications. *Proceedings of the CHI '86*, Boston (April), New York: ACM, 216-220.

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.

Tanik, Murat M., and Yeh, Raymond T. (1989) Rapid Prototyping in Software Development, guest editors for special issue of *IEEE Computer, 22*, 5 (May), 9-10.

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, 171-180.

Wasserman, A. I., and Shewmake, D. T. (1985) The Role of Prototypes in the User Software Engineering (USE) Methodology. Chapter 7 of H. Rex Hartson (Ed.), *Advances in Human-Computer Interaction, Vol. 1* Norwood, NJ: Ablex, 191-209

Weiser, Mark (1982) Scale Models and Rapid Prototyping. *ACM SIGSOFT Software Engineering Notes, 7*, 5 (December), 181-185.

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. T., and Mason, R. E. A. (1983) Information System Prototyping Techniques, Tools, and Methodologies. *Infor*, 21, 3 (August), 177-191.

Carey, Tom (1988) The Gift of Good Design Tools. Chapter 5 of H. Rex Hartson and Deborah Hix (Eds.), *Advances in Human-Computer Interaction*, *Vol. 2*. Norwood, NJ: Ablex, 159-174.

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

Whiteside, J., Bennett, J., and Holtzblatt, K. (1988) Usability Engineering: Our Experience and Evolution. *Handbook of Human-Computer Interaction*, M. Helander (ed.), Elsevier North-Holland, 791-817.

Whiteside, J., and Wixon, D. (1985) Developmental Theory as a Framework for Studying Human-Computer Interaction. Chapter 2 of H. Rex Hartson (Ed.), *Advances in Human-Computer Interaction, Vol. 1* Norwood, NJ: Ablex, 29-48.

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