User Interface Prototyping: Tools and Techniques

Pedro Szekely

USC/Information Sciences Institute
4676 Admiralty Way, Marina del Rey, CA 90292
Phone: (310) 822-1511, FAX: (310) 823-6714
szekely@isi.edu

INTRODUCTION

Prototyping is an important technique to reduce the cost and risk involved in developing complex software systems [Rudd 94]. It essentially involves building a small scale version of a complex system in order to acquire critical knowledge required to build the system. Even though prototyping involves building only a small scale version of a system, significant costs and risks are still involved. The prototyping process takes time, involves many people, and if incorrect or incomplete knowledge is gathered it can lead project managers, system builders and end-users to make false assumptions about important characteristics of a system, setting up the stage for a project failure.

In this paper we survey different tools and techniques for prototyping user interfaces, ranging from paper and pencil to draw mockups of displays to sophisticated interface construction toolkits. We view prototyping as an information gathering process, so we will compare the tools and techniques according to two criteria. The first criterion is a measure of the completeness and variety of the information that a tool or technique can help acquire, given that several different kinds of information are needed to design and build a good user interface. The second criterion is the ability of the tools to expedite the information gathering process in order to minimize the cost of the prototyping process and to maximize its effectiveness.

The paper is organized as follows. In the next section we discuss why prototyping is important in user interface development. Then we categorize the different kinds of information that need to be collected during the prototyping process, and describe a set of requirements that prototyping tools and techniques must satisfy to make interface prototyping effective. The body of the paper compares different kinds of prototyping tools and techniques according to the two criteria mentioned above. We close with a summary and an agenda of research issues.

PROTOTYPING USER INTERFACE SOFTWARE IS IMPORTANT

The conventional wisdom today is that the only way to build good user interfaces is by iterative refinement [Buxton 80]: build an initial version of the interface, and then test it with users and revise it as many times as you have money and time for [Gould 85, Swartout 82]. In a recent study of 74 software development projects in industry and academia [Myers 92a], 87% of interviewed developers reported using iterative design. In addition, some of the best examples of interactive software today were developed using iterative design: the Xerox Star [Bewely 83], the Apple Lisa and Macintosh [Morgan 83] and the Olympic Messaging system [Boies 85].

Prototyping is an invaluable technique for iterative design because iterative design involves making many revisions to the implementation of a design, and, as we argue below, building and revising the actual software system is difficult and expensive.

High quality interface software is complex. It has to work in real time, it must respond appropriately to all possible inputs, it must allow users to undo and interrupt operations, and often it must be multiprocessed so that the system will respond to users even while performing a long computation. Since interface software is complex, it is not surprising that the amount of software devoted to user interface in modern interactive systems is large. The study mentioned above also reported that on average over 50% of code for a wide set of applications is devoted to user interface. Many developers also report [Rosson 87] that they find it very hard to modularize the user interface software from the rest of the system, finding it harder to revise the software when needed.

Prototypes of interface software can ignore many of the requirements mentioned above, making the software simpler and smaller, and thus cheaper to develop and revise. For these reasons, prototypes can expedite the iterative refinement cycle required to build good user interfaces.

PRODUCTS OF THE PROTOTYPING PROCESS

The main purpose of a prototype is to allow developers to acquire the information needed to successfully build a system. We first review the kinds of information needed to develop a successful interface, and then we will compare prototyping tools and techniques in terms of their ability to maximize the effectiveness and minimize the cost of acquiring the information.

To build a successful interface developers need to acquire several kinds of information about the system to be built. The information ranges from an analysis of the tasks that users are expected to perform with the system, to detailed descriptions of the look and feel of the system. The required information falls into the following categories.

Task specification: specification of the tasks that users are expected to accomplish using the system. The task specification is needed in order for developers to understand what services to provide in a system and how to deliver them to the end-users in order to help them perform their tasks more effectively. Prototypes help developers better understand the tasks that users need to perform by letting developers see users in action with the system, and get feedback about the effectiveness of a system. In addition, successful systems often change the nature of the tasks that users perform, enabling them to do tasks they could not accomplish before. Prototypes let developers see how a system might change the tasks that users perform, and allow them to design a better system.

System functionality: specification of the functional requirements of a system. This information specifies the requirements of the software modules that the interface software calls upon to fetch data display, and to modify data in respone to user requests. Different interface designs often impose different requirements on system functionality. Tools that can build interface prototypes without requiring that the system functionality be built are especially useful. They allow interface designers to explore design alternatives without having to wait for programmers to revise the system functionality in order to test the prototypes.

Interface functionality: a specification of the system information and state that needs to be presented, and the commands to be made available to users. This specification captures the "content" of the interface, abstracting away from "style" details such as font, color, etc. It is important for developers to understand the interface at this abstract level. For example, before worrying about style issues of a display, developers should understand whether the display presents the right amount of data at the right time to help users perform their tasks.

Screen layouts and behavior: specification of how the interface looks and behaves. This information is of primary importance because it defines what users can see and do. Prototyping is very useful to acquire this information because look and feel issues are a source of endless

discussions between members of a design team, and there is a wide space of possibilities that need to be explored.

Design rationale: a specification of the reasons why the different design choices were made. This information is useful for many reasons. It can be used to achieve consistency in the interface, to guide extensions to the interface of a previous version of the system, to review and justify designs with management, etc. In addition, since prototypes often implement only a subset of the system functionality, the design rationale is useful in extrapolating from the prototype to the full system.

User feedback: a log of feedback about an interface collected from a variety of sources such as end-users, management and design reviews. Feedback can take several forms: comments user express while interacting with a prototype, answers to questionnaires, video segments, and complete interaction histories. Collecting and managing feedback is an important and difficult task. Prototyping tools should provide facilities to collect and manage such feedback so it can be retrieved when needed.

Response times: a specification of the required system response times in different situations. Prototypes let developers see the users in action and understand the required response times for effective system usage.

Reusable code: a by-product of building a prototype can be reusable code that can be used in the implementation of the real system. Building a prototype can be expensive, and the cost is harder to justify when the prototype's implementation cannot be reused in the implementation of the real system.

None of the currently available prototyping tools and techniques can deliver all the kinds of information mentioned above. In addition, the ability to capture the relevant information is not the only criteria for evaluating prototyping tools and techniques: there are other requirements.

REQUIREMENTS FOR PROTOTYPING TOOLS

Prototyping tools and techniques differ substantially in the support they provide for acquiring the various kinds of information described in the previous section. In this section we describe a set of requirements for prototyping tools in order to make the process of gathering such information effective and cheap.

Ease of use. Interface development is a team effort involving end-users, system analysts, programmers, interface designers, graphic artists, etc. Prototyping tools should allow all members to participate in the development and refinement of the prototype. Steep learning curves are unacceptable because many of the potential contributors to the prototyping process do not have the time to learn the tools. In addition, difficulty of use slows down the iteration cycle necessary to develop good user interfaces.

Fast turn-around. Interface prototyping involves making many small refinements to the interface. Tools should allow developers to quickly make the changes and immediately see the interface in action again.

Extensive control over prototype features. Prototyping tools should be very flexible. One of the purposes of prototyping tools is to try out new ideas, so prototyping tools should support a large variety of interface designs, and should give developers extensive control over design details.

Data collection capabilities. Ideally, prototyping tools should capture all the different kinds of information mentioned in the previous section.

Executable prototypes. Prototypes should be as faithful to real systems as developers need to make them in order to increase the reliability of the information collected. An executable prototype is one that can respond to user input and provide appropriate responses. However, it is not always necessary that prototypes be connected to real data and that they perform real computations; simulated data is appropriate in many situations.

Lifecycle support. Prototyping tools should help with all phases of development starting with early conceptual design through detailed screen and behavior design. In addition, prototyping can be relevant after system deployment in order to do redesing and to try out enhancements.

Team design. Software products are developed by teams. Prototyping tools should support groups of people working together either simultaneously or asynchronously, and perhaps remotely.

Version control. An important aspect of prototyping is to explore and evaluate alternative designs. Many versions of a prototype might be built while exploring different alternatives. Developers might want to revisit previous designs, so keeping and managing prototype versions is important. In addition, user feedback about a prototype should be tied to the version on which it was collected to facilitate exploring designs to address issues raised during user testing.

PROTOTYPING TOOLS AND TECHNIQUES

The essence of user interface prototyping is to construct a small scale version of an interactive system to collect information to guide its construction. Interface prototypes can be built with a large variety of tools, ranging from paper and pencil to draw mockups of displays to sophisticated interface construction toolkits. In this section we describe different categories of tools that can be used to prototype interfaces, highlighting their special strengths and weaknesses. We compare the tools with respect to their ability to deliver the different kinds of information needed to build good user interfaces, and with respect to the requirements outlined in the previous section.

Paper and Pencil

Paper and pencil are perhaps the most popular tools one uses to describe interface designs to others. Under this category we also include electronic versions of these tools such as drawing, painting and text editors.

Paper and pencil are prototyping tools with many strengths. They are easy to use: most people can draw boxes with buttons, menus and scribbles representing the objects in an application domain. Paper and pencil allows extensive control over details of the design. Control is not even limited by our ability to draw, because we can always draw a scribble and tell others what it means. Paper and pencil also encourage team design because many people can draw at the same time, especially when drawing on blackbords and large sheets of paper. Paper and pencil are also very useful for capturing different kinds of information, because whatever is not captured in the drawings can easily be expressed with textual annotations.

The main weaknesses of the paper and pencil technique is that it is very awkward to capture behavior and that the interface prototypes are not executable. Behavior is often captured using two drawings showing the interface before and after an action, together with an annotation of what the action is (e.g. clicking the mouse). Not only are such descriptions unwieldy, but since they are not executable, they capture only the "look" of the interface, but not its "feel".

Despite their weaknesses, paper and pencil, or their electronic versions, are invaluable complements to all the other prototyping tools because they complement their weaknesses (see Tables 1 and 2 in the summary section at the end of the paper).

Facade Tools

Facade tools are essentially drawing editors with an ability to specify input behavior. We call them *facade* tools because they allow developers to construct screens that look and behave like the screens of the real application, except that there is no "application" behind them. The screens display canned data, and the behaviors either switch to another canned screen, or update the screen with a new set of canned data.

Tools in this category differ mostly in the quality of the drawings (e.g. 3D shapes), and the sophistication of the input behaviors that can be specified. Examples of facade tools are Astound [Astound 93], Hypercard [Hypercard 92] and MacroMind Director [MacroMind 90]. These three tools, and other similar tools, were not designed as user interface prototyping tools. They have their own domain of applicability, and they are reputed to be widely popular and extremely effective tools in their domain. Here we analyze them as tools for prototyping interfaces, because they are often used as such.

Astound is a presentation preparation package. It allows users to produce a sequence of slides containing both text and graphics. Users can also add to a slide buttons with associated behaviors such as jumping to another slide, or making elements of the slide appear and disappear. Astound provides sophisticated animation capabilities to dramatize the transitions between slides.

Astound can be used as an interface prototyping tool. Developers can draw the displays of an application and use the buttons to show the sequence of displays that users will need to traverse to accomplish different tasks.

Hypercard is a tool to build hypertext applications. Hypercard applications are built with two kinds of abstractions, cards and stacks of cards. Cards can contain a variety of fields and pictures. The fields can be type-in areas, buttons, menus and other WIMP interface building blocks (i.e., interfaces consisting of Windows, Icons, Menus and Pointing). Hypercard has three modes of operation: end-user, simple authoring and application developer. The end-user mode allows the user to interact with the Hypercard application, but not to extend it in any way. The simple authoring mode allows the user to insert links between cards, change layouts and edit scripts. The application developer mode gives developers full access to all Hypercard capabilities, including the ability to define new cards and new fields, and to define and modify scripts.

Hypercard's simple authoring and developer modes provide excellent facilities for prototyping interfaces. The simple authoring mode allows all members of the design team to be involved in screen design and simple behavior definition. Developers with programming skills can additionally use the developer mode to write scripts to implement complex behaviors to produce highly functional prototypes. End-user mode allows users and developers to test the interface.

(a) (b)

Figure 1. Initially, creating detailed graphics (a) may distract the team from fundamental questions, and cause them to dwell on details, such as the lighting model applied. (b) allows designers to address higher level issues, such as the action pressing this button will cause[1].

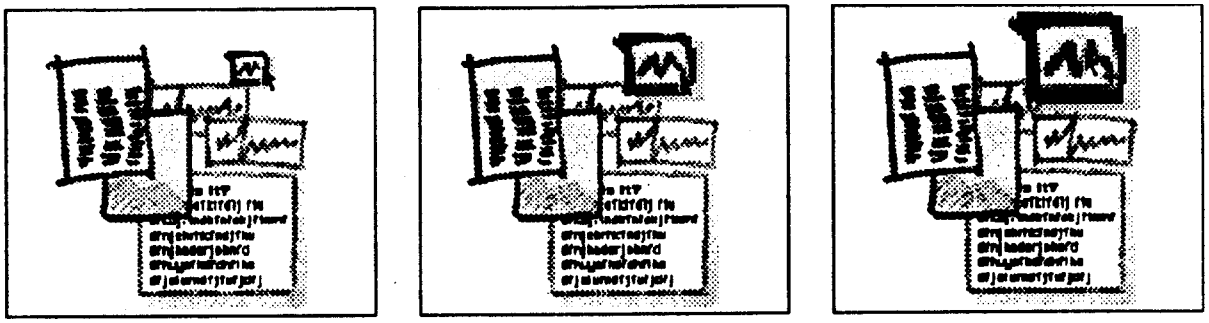

Figure 2. The sequence shows (albeit in static form) how a user might interact with a proposed system. The user drags a movie (box with the 'M' in it) to a group of diverse documents, while controlling the resolution of the movie by zooming it up or down. The rough sketch avoids details, and allows designers to focus on issues such as : Will users want this capability? Is this a collection a group? How should the user specify moving an individual item versus the whole group? What does it mean to zoom a movie up and down? Should the movie play as it is moved or zoomed? etc[1].

MacroMind Director is a sophisticated multimedia authoring tool. It supports 3D graphics, large libraries of animation effects, and a sophisticated scripting language. MacroMind Director is especially useful for prototyping interface with a large graphical component, unlike Hypercard which is useful mainly for WIMP interfaces. Figures 1. and 2. illustrate the power of MacroMind Director for prototyping interfaces at the interface functionality level rather than at the look and feel level.

Facade tools retain most of the benefits of paper and pencil, while adding the ability to describe behavior and to execute the prototypes. Many facade tools can give end-users the illusion of interacting with the real application, so they can be used to get very reliable feedback from end-users.

The main weakness of facade tools appears to be that they do not produce reusable code that can be used to build the real application, so the implementation effort in building the prototype is lost. Additionally, since the prototyping and the implementation tools are so different, the prototype and the application might be built by different teams of people, and many of the lessons learned while building the prototype remain in the minds of the prototyping team, and do not carry over effectively to the implementation of the system.

---

[1]Figure taken from Yin Yin Wong's paper "Rough and Ready Prototypes: Lessons from Graphic Design" in CHI'92, Posters and Short Talks. May, 1992.

On the other hand, as Brooks [Brooks 79] argues, developers should always plan on throwing away the first implementation of a system, which means that the main use of the first implementation of a system is to learn how to build the system correctly the second time. Facade tools could be viewed as a way of acquiring all the relevant knowledge without having to actually implement the system, so the amount of work thrown away is smaller. Unfortunately, there is not enough evidence in the literature to decide whether the lack of reusable code is really a serious shortcoming.

A potential problem of facade tools is that they can over-sell the capabilities of an application giving the illusion that a more sophisticated application will be constructed than what is feasible given budget, time and technology constraints.

Interface Builders

Interface builders, unlike facade tools, are interface *construction* tools rather than interface prototyping tools. Their main strength is that, like facade tools, they give interface developers a drawing-like interface to specify the interface, but, unlike facade tools, generate executable code that can be linked in into an application to produce an industrial strength implementation. While facade tools provide a special scripting language to specify behavior, interface builders use a general purpose programming language such as C or C++ to specify behavior, the same language that is often used to implement the application functionality.

There are dozens of interface builder tools in the market such as the NeXT Interface Builder for NeXT Step [NeXT 91], Prototyper for the Macintosh [SmethersBarnes 90] , WindowsMAKER for Microsoft Windows [BlueSky 91] and UIMX for X Windows and Motif [VisualEdge 90]. A recent study by Myers [Myers 92a] revealed that interface builders are widely used and are reported to greatly facilitate the interface construction process.

Interface builders are often marketed as prototyping tools because they satisfy many of the requirements of prototyping tools that were mentioned above. They are easy to use because they provide a drawing-style interface for building displays. They provide fast turnaround to changes because of their drawing-like interface and their ability to quickly switch between "build" mode where developers specify the elements of displays and "run" mode where developers can test the interface as if they were end-users. They provide extensive control over interface designs because developers can easily change all the properties of the elements of a display such as layout, fonts, colors, etc., and, of course, the prototypes are executable.

Interface builders have three major shortcomings when used as prototyping tools. First, they can only be used to construct the static portions of an interface such as the menus and dialogue boxes that control the application. Interface builders cannot be used to specify the "main windows" of applications, which display application-specific information in graphical ways (e.g., shapes in a drawing editor, circuit elements in a schemaitcs editor, notes in a music editor), and that allow users to directly manipulate the information. Interface builders only allow developers to reserve an area where this information will be displayed, but its contents must be programmed using the primitives of the underlying toolkit and window system. The need to program the interface to the main application window makes interface builders break down as prototyping tools. Ease of use is lost because only expert programmers rather than graphics artists, interface experts, domain experts and end-users have the sophisticated programming skills required to build these windows. Fast turn around to design changes is lost because programming is a slow process.

The second major shortcoming of interface builders is that it is difficult to isolate the interface from the rest of the application. Interface builders require programmers to write a large number of procedures that are automatically called when the interface elements are activated by end-users (there procedures are called call-backs). Changing the interface requires rewriting callback procedures in order to keep the interface executable, eliminating ease of use and fast turn around.

In addition, interface builders provide little support for simulating portions of an application that have not yet been implemented. Programmers must write call-back procedures that insert the simulated information into the appropriate interface elements.

The third shortcoming of interface builders is that they require developers to select concrete building blocks to specify interfaces (sliders, buttons, etc.) forcing them to commit to specific interface features before they are ready to do so. For example, consider a designer who wants to specify that a microwave oven interface should have a feature to select the cooking temperature. An interface builder forces the designer to specify exactly how the temperature will be selected (e.g., slider, numeric pad, menu) even though the designer is not ready to choose a specific technique. This makes interface builders useful for producing screen layouts and behavior, but not for producing information about interface functionality, which as was mentioned before, is critical for designing good interfaces.

Model-Based Tools

Model-based interface development is a new paradigm for developing interfaces [Foley 91, Puerta 93, Szekely 92, Wiecha 90]. The model-based paradigm uses a central database to store a description of all aspects of an interface design (Figure 4). This central description is called a *model*, and typically contains information about the tasks that users are expected to perform using the application, the data of the application, the commands that users can perform, the presentation and behavior of the interface, and the characteristics of users.

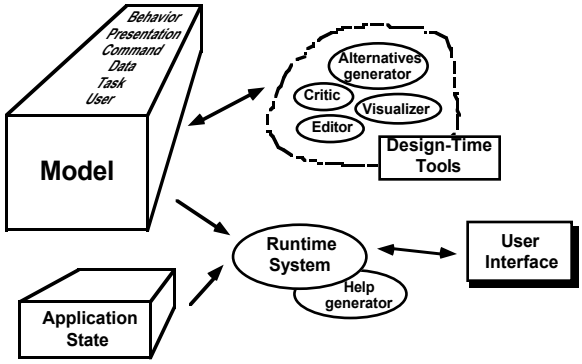

Figure 4. Architecture of model-based interface development tools.

A standard software module called an *interface generator*, or *runtime system* uses the model as input, and maps the state of the application into the windows that appear on a user's screen. The runtime system also accepts inputs from the user and invokes the appropriate commands.

Interfaces are developed by using specialized design-time tools to build and refine models. Developers specify *what* features the interface should have, rather than write programs that specify *how* to make the computer exhibit the desired behavior.

The main advantage of the model-based approach over traditional interface development approaches is that it enables the construction and use of tools to provide assistance to interface developers. For example, UIDE [Foley 88, Foley 91] has design critics that automatically detect a variety of inconsistencies in an interface design and evaluate designs using GOMS analysis [Kieras]85; Humanoid [Luo 93, Szekely 92, Szekely 93] provides model editors and visualizers, alternatives generators, and interface generators for creating prototypes of a design, even before the design is complete (Figure 5).

Model-based tools blur the distinction between prototyping and implementation. The models used in the model-based tools have place-holders for capturing interface designs at several levels

of abstraction, making it possible to capture many of the products of the prototyping process such as task specifications, system functionality, interface functionality, screen layout and behavior, and several kinds of design rationale. This same knowledge is used by the run-time system to implement the interfaces.

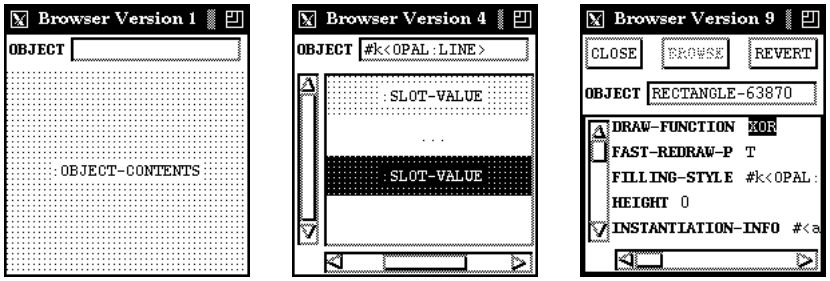

Figure 5. The sequence shows snapshots of an interface prototype for a browser application at different stages of development. The prototype was developed using Humanoid [Szekely 92]. In the first snapshot, the developer has specified a type-in area for specifying the object to be browsed, and an area called OBJECT-CONTENTS, where the contents of the object will be shown. The area is shaded to indicate that the presentation of the object's contents has not been specified. The second snapshot shows the interface after a few refinement steps. The developer specified that the contents of the object should be shown in a scrolling area and that it consists of a list of slot/value pairs, whose presentation is not yet specified. However, the developer specified that the slot/value pairs should be selectable by clicking the mouse. The prototype allows the slot/value pairs to be selected, and highlights the selected element, even though the presentation is not yet specified. The last snapshot shows the completed interface.

Model-based tools also satsify many of the requirements we laid out for prototyping tools. They provide moderate control over the design, allowing developers to express designs at different levels of abstraction. The run-time system can prototype the interface before the design is complete, providing fast turn around for design changes, and allowing end-users to interact with the prototype before the interface is complete. For example, in Humanoid, the unspecified portions of the design are displayed as dotted areas to indicate that they are not fully specified (Figure 5.). Given that designs can be specified at several levels of abstraction, model-based tools support prototyping throughout the lifecycle of a system, from the initial conceptual design stages to the detailed refinement of the implementation and maintenance stages.

Even though some model-based tools such as Humanoid provide interactive interfaces for model-building, prototyping an interface using these tools is not as easy as with facade tools or interface builders. Model-building involves describing interfaces declaratively, which is not as easy as drawing pictures. However, model building is easier than programming. Tools for model-building are being actively researched, and we expect to see substantial improvements with tools using demonstrational techniques [Myers 87, Myers 89, Myers 92b].

Model-based tools have three main weaknesses. First, they are difficult to use compared to interface builders, facade tools and paper and pencil, even though current research is addressing ease of use issues very actively [Myers 92c, Szekely 93]. Second, they provide only moderate control over the details of highly graphical user interfaces, and third, since the model-based technology is not mature, the tools are not sufficiently efficient and reliable for widespread use.

Domain-Specific Tools

Domain-specific tools are tools for building special kinds of applications (e.g., database applications) or applications with specific styles of interfaces (e.g., Hypercard). By focusing on a narrower domain, these tools can provide powerful facilities for constructing applications very quickly. Here we look at these tools as prototyping tools because the effort for building

applications is often so small that it compares to the effort needed to build prototypes using the other prototyping techniques discussed in this paper.

Fourth-generation languages (4GLs) are special purpose programming languages for constructing database applications [cite]. The language provides facilities for defining the database schemas, facilities for querying and updating the database, and facilities for defining forms to allow end-users to query and update the database. The form definition language allows different kinds of fields to be placed on the screen (e.g., text, numeric, choice), and provides special support for validating inputs and triggering procedures when the fields are changed.

4GLs are powerful tools. Once the database schemas are defined, most of the work centers around defining the forms for querying and updating the information in the database. For example, a database application to store medical histories and billing records for patients in a private doctor's practice took two weeks to develop. The application features four different screens of information, supporting different views of the information.

Even though 4GLs are system implementation tools, they satisfy many of the requirements of prototyping tools outlined in previous sections. 4GLs provide fast turn around for changes and provide extensive control over details of a design, within the constraints of the database application domain. The prototypes are, of course, executable.

The main weakness of 4GLs within their narrow domain of applicability is execution speed. Many 4GLs are claimed to be too slow for large scale applications. Thus, rather than considering 4GLs as implementation languages with strong prototyping features, one might consider 4GLs to be prototyping tools with enough functionality that the prototype can often serve as the final system implementation. Compared to other prototyping tools, 4GLs are hard to use, even though as implementation tools they are reasonably easy to use. 4GLs require training, but can easily be learned by people without a formal computer science training.

Hypercard is another example of a narrow domain tool. In a previous section we discussed Hypercard from the point of view of a facade tool and found it to be a good interface prototyping tool, its main weakness being that it does not produce reusable code. However, for applications fitting Hypercard's "card" paradigm, Hypercard can be an implementation rather than a prototyping tool. In these cases, Hypercard's main weakness as a prototyping tool goes away: Hypercard's scripting language becomes the implementation language, and prototype and implementation become indistinguishable. Like 4GLs, Hypercard's main weakness is execution speed and narrow domain of applicability.

Visual Basic [VisualBasic 93] is an application building environment with many features of interface builders, Hypercard and 4GLs. Visual Basic can be viewed as an interface builder that uses Basic as the programming language for writing the call-back procedures. The use of Basic is significant because Basic is an easy to learn interpreted programming language, addressing the more serious prototyping shortcomings of interface builders: difficulty and slow turn-around for prototyping behavior and interfaces for application-specific information. Visual Basic can also be viewed as a Hypercard-like tool where the notion of stacks is relaxed. They both use an easy to learn scripting language, but VisualBasic supports applications with heterogeneous sets of windows rather than applications with a stack of similar looking windows. Also, VisualBasic offers more control over the interface of the application being prototyped since it does not impose a default interface for browsing a stack of cards (e.g., commands to go to the next and previous cards, etc.). VisualBasic can also be viewed as a 4GL given its connectivity to databases and its facilities to build the windows to interact with the information. The main advantage of VisualBasic over 4GLs are its powerful interface builder-like capabilities, which are not found in most 4GL tools.

Products of the prototyping process						
	Paper & Pencil	**Facade Tools**	**Interface Builders**	**Model-based Tools**	**Domain-specific Tools**	**Actual Implementation**
Task spec.	Yes	No	No	Yes	No	No
System functionality	Yes	No	No	Yes	Yes	Yes
Interface functionality	Yes	Implicit	Implicit	Yes	Implicit	Implicit
Screen Layout & Behavior	Yes	Yes	Yes (static) No (dynamic)	Yes	Yes	Yes
Design Rationale	Yes	No	No	Partially	No	No
User Feedback Record	Yes	No	No	No	No	No
Response Times	No	Yes	Yes	Yes	Yes	Yes
Reusable Code	No	No	Yes	Yes	Yes	Yes

Table 1. The table shows the ability of different classes of prototyping tools and techniques to help developers collect different kinds of information needed for effective user interface development.

Actual Implementation

We finish our survey of interface prototyping tools by discussing the strengths and weaknesses of using the actual implementation of a system as its own prototype. Given that interface software is complex and large, there are not many benefits in using the actual implementation of a system for prototyping purposes, except for the case of domain-specific tools discussed in the previous section. The actual implementation of a system fails to meet the most fundamental requirements of prototyping tools: ease of use allowing participation by graphic artists, domain experts, end-users and other experts lacking programming expertise, and slow turnaround to changes.

SUMMARY

Tables 1 and 2 summarize the strengths and weaknesses of the different prototyping techniques discussed in the previous sections. Table 1 compares prototyping techniques with respect to their ability to deliver the products expected from the prototyping process. Table 2 compares the techniques from the point of view of requirements for supporting the prototyping process.

Table 1. shows that paper and pencil, or an electronic version of them, are an invaluable complement to all other prototyping techniques. Paper and pencil can be used to record all kinds of information that cannot be captured using the other tools. From the point of view of capturing the products of the prototyping process, the model-based tools are the best. However, as shown in Table 2. model-based tools do not satisfy the requirements to support the prototyping process as effectively as some of the others.

Prototyping tool requirements						
	Paper & Pencil	**Facade Tools**	**Interface Builders**	**Model-based Tools**	**Domain-specific Tools**	**Actual Implementation**
Ease of Use	Excellent	Good	Good	Fair	Good, moderate	Poor
Fast Turnaround	Excellent	Excellent	Good	Excellent	Good	Poor
Extensive Control	Excellent	Excellent	Good	Moderate	Fair Good within domain	Excellent
Data Collection	Good	Fair	Fair	Excellent	Good	Good
Executable Prototypes	None	Moderate	Good	Excellent	Excellent	Excellent
Lifecycle Support	Good	Moderate	Moderate	Good	Moderate	Poor
Team Design	Good	Poor	Poor	Poor	Poor	Fair
Version Control	Poor	Poor	Poor	Poor	Poor	Fair

Table 2. The table shows the extent to which different classes of prototyping tools and techniques satisfy a variety of requirements for effective interface prototyping.

Table 2. shows that there is no single technique or class of tools that is uniformly better at satisfying all the requirements of the prototyping process. Paper and pencil are excellent in terms of ease of use, fast turnaround and extensive control over details. However, the impossibility to execute the prototypes limits their applicability to the early stages of the design. Facade tools are the next best in terms of the first three requirements, but as Table 1. shows, they cannot produce reusable code, a serious shortcoming. Interface builders satisfy the requirements of the prototyping process fairly well, but their lack of support for prototyping behavior limits their applicability to only some aspects of the complete interface. The domain-specific tools, when applicable, are the most effective tools.

RESEARCH ISSUES

Tables 1. and 2. show that interface tools and techniques can still be improved substantially.

Model-based tools are making substantial progress in capturing the different types of information needed for construcing interfaces. However, the tables suggest that a good complement to all other tools would be a mechanism for annotating the formal objects of the tools with arbitrary text and graphics. Such annotations would be like the yellow PostIt™ notes often used in office environments. They would be used to capture information that cannot be captured in the specific tools, and in addition, remain attached to the objects of interest.

Ease of use is a big concern. Currently, paper and pencil, and facade tools are the easiest to use for prototyping complex interface features. We expect the popularity of these tools to decrease as the flexibility and ease of use of tools that can produce reusable code increases. For example, demonstrational techniques are being used in Garnet [Myers 92c] and Druid [Singh 90] to allow designers to specify more complex behavior and application-specific displays with the same ease with which interface builders let designers specify the static portions of a display.

Model-based tools offer a lot of promise for both interface prototyping and implementation. However, further research is needed to bring them into the main-stream. None of the existing model-based tools have the capability of modeling arbitrary interfaces, and the algorithms for

interpreting the models to produce the interfaces are slow compared with the special purpose procedural implementations used today.

Support for team-design and version control are lacking from most of the tools discussed in this paper. Only paper and pencil tools provide some support for team design. Pictive and TelePICTIVE [Miller 92] are examples of paper and pencil tools which address team design as one of their mayor concerns.

REFERENCES

Astound 93 Astound User's Guide. Gold Disk Inc. P.O. Box 789, Streetsville, Mississauga, Ontario, Canada L5M 2C2.

Bewely 83 W. L. Bewely, T. L. Roberts, D. Schroit and W. L. Verplank. Human Factors Testing in the Design of Xerox's 8010 'Star' Office Workstation. In Human Factors in Computing Systems, pages 72-77. Proceedings SIGCHI'83, Boston, MA, December, 1983.

BlueSky 91 WindowsMAKER. Blue Sky Software Corporation, 2375 East Tropicana Ave., Suite 320, Las Vegas, NV 89119. Phone (702) 465-6365, 1991.

Boies 85 S. J. Boies, J. D. Gould, S. Levy, J. T. Richards and J. W. Schoonard. The 1984 Olympic Message System—A Case Study in System Design. IBM Research Report RC-1138, 1985.

Brooks 79 F. P. Brooks. The Mythical Man-Month: Essays on Software Engineering. Addison-Wesley, 1979.

Buxton 80 W. Buxton and R. Sniderman. Iteration in the Design of the Human-Computer Interface. In Proceedings of the 13th Annual Meeting of the Human Factors Association of Canada. 1980, pp. 72-80.

Foley 88 J. Foley, C. Gibbs, W. Kim, and S. Kovacevic, A Knowledge Base for a User Interface Management System, Proceedings CHI '88 - 1988 SIGCHI Computer-Human Interaction Conference, ACM, New York, 1988, pp. 67-72.

Foley 91 J. Foley, W. Kim, S. Kovacevic and K. Murray, UIDE - An Intelligent User Interface Design Environment, in J. Sullivan and S. Tyler (eds.) Architectures forIntelligent User Interfaces: Elements and Prototypes, Addison-Wesley, Reading MA, 1991, pp.339-384.

Gould 85 J. D. Gould and C. H. Lewis. Designing for Usability - Key Principles and What Designers Think. Communications of the ACM 28(3):300-311, March, 1985.

Hypercard 92 Claris Corporation, 1992.

Kieras 85 D. E. Kieras and P. G. Polson. An Approach to the Formal Analisys of User Complexity. International Journal of Man Machine Studies, 22, 365-394.

Luo 93 P. Luo, P. Szekely and R. Neches: Management of interface design in HUMANOID. In Proceedings INTERCHI'93. April 93.

MacroMind 90 MacroMind Director. MacroMind, 410 Townsend, Suite 408, San Francisco, CA 94107. Phone (415) 442-0200. 1990.

Mahler 90 P. Mahler. An Informix-4GL Tutorial. Prentice Hall, Englewood Cliffs, NJ 07632, 1990.

Miller 92 D. S. Miler, J. G. Smith and M. J. Muller. TelePICTIVE: Computer-Supported Collaborative GUI Design for Designers with Diverse Expertise. In UIST'92 Proceedings, 1992, pp. 151-160.

Morgan 83 C. Morgan, G. Wiliams and P. Lemmons. An Interview with Wayne Rosig, Bruce Daniels and Larry Tesler. Byte 8(2):90-114, February, 1983

Myers 87 B. A. Myers. Creating Dynamic Interaction Techniques by Demonstration. Proceedings of ACM CHI+GI'87 Conference on Human Factors in Computing Systems and Graphics Interface. 1987. pp.271-178.

Myers 89 B. A. Myers, B. Vander Zanden and R. B. Dannenberg. Creating Graphical Interactive Application Objects by Demonstration. Proceedings of ACM SIGGRAPH 1989 Symposium on User Interface Software and Technology (UIST '89), 1989, pp.95-104.

Myers 92a B. A. Myers and M. B. Rosson. Survey on user interface programming. In Proceedings of CHI'92, The National Conference on Computer-Human Interaction, May, 1992, pp. 195-202.

Myers 92b B. A. Myers. State of the Art in User Interface Software Tools. In H. Rex Hartson and Deborah Hix, Ed. , Advances in Human-Computer Interaction, Volume 4, Ablex Publishing, 1992.

Myers 92c B. Myers, et. al. The Garnet Reference Manuals. Technical Report CMU-CS-90-117-R2, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213. May 1992.

NeXT 91 NeXTStep and the neXT Interface Builder. NeXT, Inc. 900 Chesapeake Drive, Redwood City, CA 94063. 1991.

Puerta 93 A. Puerta. The Study of Models of Intelligent Interfaces. In Proceedings of the ACM International Workshop on Intelligent User Interfaces. Jan, 1993. pp. 71-78.

Rosson 87 M. B. Rosson, S. Maass and W. A. Kellogg. Designing for Designers: An Analysis of Design Practices in the Real World. In Human Factors in Computing Systems, pp. 137-142. CHI+GI'87, Toronto, Ont., Canada, April, 1987.

Rudd 94 J. Rudd and S. Isensee. Twenty-Two Tips for a Happier, Healthier Prototype. ACM Interactions 1(1):35-41. January, 1994.

Singh 90 G. Singh, C. H. Kok, and T. Y. Ngan. Druid: A system for Demonstrational Rapid User Interface Development. Proceedings of ACM SIGGRAPH 1990 Symposium on User Interface Software and Technology (UIST '90), 1990, pp.167-177.

SmethersBarnes 90 Prototyper 3.0. SmethersBarnes, P. O. Box 639, Portland, Oregon 97207, Phone (503) 274-7179, 1990.

Swartout 82 W. Swartout and R. Balzer. The Inevitable Intertwining of Specification and Implementation. Communications of the ACM 25(7):438-440, July, 1982.

Szekely 92 P. Szekely, P, Luo, and R. Neches. Facilitating the Exploration of Interface Design Alternatives: The HUMANOID Model of Interface Design. In Proceedings SIGCHI'92. May 1992, pp. 507-515.

Szekely 93 P. Szekely, P. Luo, and R. Neches. Beyond Interface Builders: Model-Based Interface Tools. In Proceedings of INTERCHI'93 April, 1993, pp. 383-390.

VisualBasic 93 Visual bassic Programmer's Guide. Microsoft Corporation, One Microsoft Way, Redmond, WA 98052-6399.

VisualEdge 90 UIMX. Visual Edge Software Ltd., 3870 Cote Vertu, Montreal, Quebec, Canada H4R 1V4. Phone (514) 332-6430, 1990.

Wiecha 90 C. Wiecha, W. Bennett, S. Boies, J. Gould and S. Greene. ITS: A Tool For Rapidly Developing Interactive Applications. ACM Transactions on Information Systems 8(3), July 1990. pp. 204-236.

Wong Y. Y. Wong. Rough and Ready Prototypes: Lessons form Graphic Design. in CHI'92, Posters and Short Talks. May, 1992.