Splitting the Difference: Exploring the Middle Ground in User Interface Design

Sarah Douglas, Eckehard Doerry & David Novick

CIS-TR-90-12
June 1, 1990

DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE
UNIVERSITY OF OREGON

An early, condensed version of this paper will appear in the *Proceedings of the Third Annual Symposium on User Interface Software and Technology*. A revised version of this paper will be published in the *24th Hawaii International Conference on System Sciences*.

Splitting the Difference: Exploring the Middle Ground in User Interface Design†

Sarah Douglas, Eckehard Doerry & David Novick

Department of Computer Science
University of Oregon
Eugene, OR. 97403
(503) 346-3974
douglas@cs.uoregon.edu

Abstract

The concept of interface construction kits has become become firmly established and well-accepted in recent years. Through both their popularity and the perceptions of their users, they seem to offer solutions to the programming complexity created by introducing user interface operations into traditional programming languages. We argue in this paper that an understanding of the productivity gain afforded by construction kits must be shaped by separating programming language from programming environment.

We also argue that abstraction and the creation of high-level languages, particularly object-oriented ones, reduce programming complexity. We note that when languages strive for abstraction they also strive for generality, thus losing programming power and expressiveness. Our own research, the QUICK user interface construction kit extends this research by exploring the middle ground in language abstraction. Our use of the prototype model of object-oriented languages rather than the class-based system is a relatively unique contribution.

We also describe characteristics of user interface construction kit programming environments making a distinction between weak direct manipulation environments and strong ones. Weak environments essentially support the program text as the object of interest and allow text object, menu and dialog box selection of programming language components. Structure editors are an elaborate version of this. QUICK is definitely a weak direct manipulation environment. Strong direct manipulation environments feature elements of visual programming and programming by demonstration. QUICK is also a strong direct manipulation environment, allowing the programmer to continuously display the product user interface and directly specify programming constructs by positioning and movement of those objects. We also note that

† An early, condensed version of this paper will appear in the *Proceedings of the Third Annual Symposium on User Interface Software and Technology*. This paper has been submitted to the *24th Hawaii International Conference on System Sciences*.

some programming environments allowed rapid switches between programming and testing. This supports the concept of rapid prototyping in design. QUICK is one such system.

1.0 Introduction: Complexity in User Interface Programming

With the advent of bit-mapped displays and mice, programmers are faced with ever increasing design complexity as graphics and sound supplement text displays, windows allow multiple contexts for user tasks, and pointing devices join keyboards. On sophisticated systems such as Interlisp on the Xerox 1100, the interface programming effort has been informally estimated to consume about 80% of the total programming time (Smith, 1984). The long delay experienced in software development for even less complex machines such as the Macintosh can similarly be attributed to the complexity of composing over 600 ROM-based interface functions into useable interactive programs.

The basic problem, then, is that user interface programming is becoming ever more complex. Therefore, the purpose of any interface design aid is to somehow control that complexity. One solution, the concept of an interface construction kit has received considerable attention in recent years for several reasons: A toolkit allows interfaces to be constructed with relative ease, abstracting the interface construction task and providing an efficient means of exploring large numbers of interface designs by exploiting rapid prototyping. The commercial applications of such toolkits are obvious.

While many researchers and commercial developers have focused attention on the need and feasibility of a construction kit approach to user interfaces (cf. Fischer & Lemke, 1987), these discussions have often failed to distinguish between innovations in the programming environment and those in the underlying representation of the user interface. We feel that, while innovations in either area can help to streamline interface design, they are conceptually different and each support design in distinct ways.

In this paper, we work to establish a framework for understanding complexity in user interface design tools by characterizing the domain along two orthogonal dimensions, **Language** and **Programming Environment**. The nature of the interface language establishes the conceptual and ontological framework of a user interface design tool and plays the most prominent role in determining the complexity of interface design. The programming environment provided by the design tool plays a crucial supporting role, often helping to maintain an abstraction barrier imposed by the language. In the following sections we will dissect these issues in detail, motivating the discussion by presenting our own research efforts which are illustrated in the QUICK (Quick User Interface Construction Kit) system.

2.0 Language Issues

In the past, when string oriented textual interfaces predominated, interface programming was trivial: computer languages included a handful of "I/O" functions and procedures to allow the machine to accept, process, and produce from simple data types such as characters, integers, etc. The read and write line operations of PASCAL are good

examples. However, this equation changed drastically with the introduction of complex graphical interfaces with pointing devices. As Mary Shaw (1986) has so eloquently pointed out, this signals the move from language constructs oriented towards program state to those oriented towards I/O state. In modern languages, the number of functions and procedures having to do with the interface has grown enormously to become, in some cases, the largest class of operations in the language. The Macintosh Toolbox with over 600 functions and procedures was cited earlier. Similarly, Figure 1 gives an example of this complexity for the InterLisp D language: determining whether the cursor is in a particular graphic region or not. Thus, the complexity of interface programming, associated with indexing, selecting, and combining statements from this enormous pool of low-level functions, has increased exponentially.

```
(SETQ WorkWindow
        (CreateWindow 205 307 185 295 2))
(while (InRegionP (MouseCoords)
                        (fetch ImageRegion
                                        AndGateDescr)
        and not (KEYDOWNP 'LSHIFT)
        do
                (replace CurrentCursorCoords
        (MouseCoords))
        (if (EQ (BUTTONSTATE) 'LEFT) then
                        (RETFROM 'Tracker]
```

Figure 1: Typical User Interface Code

2.1 Abstraction & Programming Complexity

The answer to this problem, as has traditionally been the case in programming languages, is abstraction. By introducing a conceptual framework specifically oriented towards interface programming, the myriad low-level functions can be composed into more abstract functional units, thereby reducing the complexity of the design space. This attractive feature is the fundamental motivation behind the concept of all high-level languages. We can conceptualize these relationships in Figure 2.

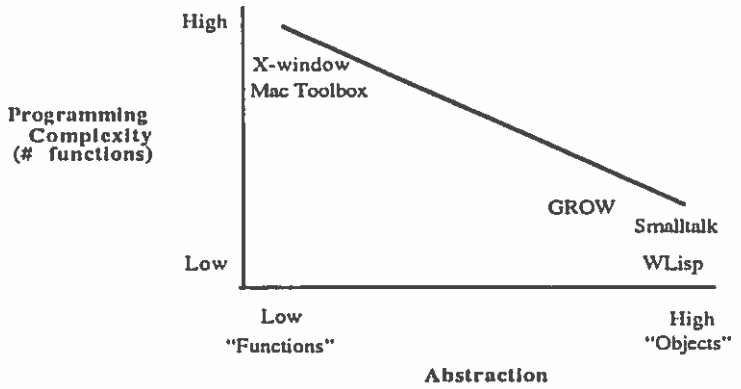


Figure 2: Abstraction versus complexity

In this figure, we place general programming languages with their large number of fine grained interface programming primitives at one end of the spectrum. For example, X-window and the Quickdraw primitives from the Macintosh toolbox appear in the low end of the spectrum. We call languages at this extreme **low-level** languages. In general, low-level languages are characterized by a lack of direct support of any ontological structure specific to interface programming. At the other end of the spectrum, we define **high-level** languages, which exibit a highly structure ontology in which the primitives are complex interface entities with concretely defined behaviors. For example, Smalltalk has complex interface objects such as windows, menus and browsers. High-level languages are typically **object-oriented** languages. Thus, it appears as no coincidence to us that current trends in managing complexity in user interface programming have directed more and more effort to object-oriented language developments.

To illustrate: Several systems have been developed specifically as object-oriented construction kits for interface design. We initially began work on this research with a great deal of inspiration from an system called *Programming by Rehearsal* (Gould & Finzer, 1984). *Programming by Rehearsal* uses a subset of Smalltalk classes with which the designer must begin. Gould and Finzer also introduce and develop the programming by demonstration paradigm. Trillium (Henderson, 1986) written in LOOPS is limited to the design of copiers but uses object composition. Fischer and Lemke's WLISP (1988) is a construction kit for windows based on an object-oriented class inheritance Lisp called ObjTalk. Fischer and Lemke's FRAMER (1988) adds design critics to the WLISP construction kit and has more of a direct manipulation environment for programming. Barth's GROW (1976) is an inheritance object-oriented system for creating graphic displays as objects (grid, text, box, icon). It supports both composition of objects and dependency between object attributes. It does not have a direct manipulation interface and does not integrate user actions. Although Cardelli's (1987) direct manipulation interface designer is not explicitly object-oriented, its dialog box components act like objects. Myers' (1987) Peridot combines programming by demonstration with active value objects.

A large number of commercial interface construction applications have evolved from these research efforts. While it would be impossible to review them all, several stand out as landmarks. Prototyper (SmethersBarnes, 1989) represents the stereotypical interface construction kit. The designer is presented with a small fixed palette of standard interface entities (eg. windows, buttons, etc.) from which to choose components of the interface. There is no abstract interface language to specify how the entities interact. Instead, the user must specify **all** interface behavior by extending the code generated by the system using a

low-level language, either PASCAL or C. Hypercard (Apple, 1987) overcomes some of the rigidity inherent in Prototyper-like kits by recognizing that a higher-level interface programming language is desirable to allow the user to easily describe basic interface functionality. This language is called Hypertalk. However, this toolkit still constrains the designer by providing a small set of predefined interface entities. Serius89 (Serius, 1989) takes this notion to an extreme: It attempts to overcome the flexibility constraints imposed by providing only predefined entities by simply giving the user an enormous set of entities to choose from. While this undoubtedly does allow a wider range of interfaces to be constructed, the complexity of indexing, selecting and combining the appropriate entities to implement a particular interface is nothing short of overwhelming. Thus, we feel that systems like Serius89 represent a "brute force" approach to the flexibility problem analogous to the PL1 language of the 60's. By focusing on simple aggregation instead of abstraction and careful design, they produce a language too large and complex to be truly useful.

2.2 Abstraction & Generality

The issue of programming complexity is not the only one that must be considered in language design. Generality must also be taken into consideration. If a language prepackages interface behaviors as objects (that is, behavior is abstracted to the point where a particular functional unit, or interface entity, embodies exactly one set of very specific behaviors), those objects must be generally useful to most language users since their overhead is expensive. Thus, most object-oriented languages and interface toolkits support only the most common interface objects, windows, menus, icons, buttons, etc., in a library usually organized in a hierarchical taxonomy from which the user may select to create new interfaces. This leaves the rest of the interface, in particular the *contents* of the various windows, without any abstraction support. If the designer is happy with this constrained rigidity, then it clearly reduces programming to the simple task of identifying the entity that embodies the behavior. On the other hand, if the designer wants to modify the behavior, then he or she is forced to descend to the composition of single functions and procedures. In other words, to retreat to using low-level language. Figure 3 illustrates that generality becomes more important as one moves towards reducing programming complexity through abstraction.

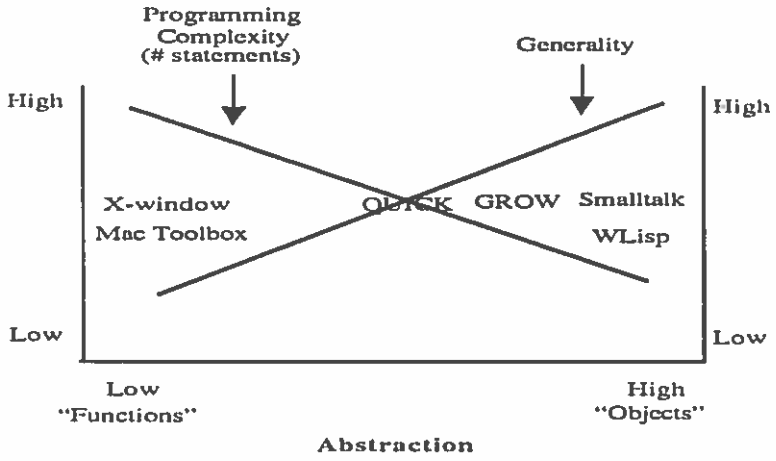


Figure 3: Abstraction versus complexity & generality

In sum, **both** ends of the spectrum have positive aspects: low-level languages offer maximum expressive power while high-level, object-oriented languages offer abstraction and efficient (though very constrained) interface construction. However, observe that there is a middle ground at the cross over point of Figure 3. In this area, programming complexity is reduced by introducing some abstraction while at the same time maintaining expressive power.

This observation is the prime motivation for our interest in this largely unexplored central area. We claim that systems in this area, which we call the **median** level, are extremely promising. They offer both structure and flexibility, and support the construction of a wide range of interfaces while requiring designers to understand only a very simple ontology and a compact high level interface programming language. By settling on an intermediate level of both ontological and functional abstraction, interface construction at the median level becomes a mixture of composition and programming: Abstract prototypical objects are *elaborated* (programmed) to exibit a particular behavior and are then composed to create the interface as a whole.

2.3 The QUICK Difference

Our QUICK system is the result of our exploration of this unique central area of the language spectrum. In particular, the QUICK language has several important features:

Simple Ontology: the Prototype Object.

A major distinction between QUICK and other approaches to interface design lies in our more abstract and flexible view of interface components. As noted previously, most toolkits provide a predefined set of interface entities (e.g. windows, buttons, etc.), each with certain behavior and attributes inherently associated with it. That is, the definition of a particular class of entity, say, a *button*, tightly constrains how instances of the class look (i.e. their screen representation), what kinds of interface events they respond to, and the nature of that response. Thus, an instance of the *button* class might be able to respond to single-clicking but not to dragging, double-clicking, or arbitrary user-defined events. Under this approach, the interface designer *chooses* interface objects which provide the desired functionality from the predefined set and *instantiates* them. Constructing an interface is therefore primarily an *instantiation* task. Unfortunately, this approach severely constrains[1] interface design: one can only build interfaces specifically supported by the predefined abstractions. In QUICK, we explore a less structured approach. Unlike most object-oriented languages, QUICK's objects are prototypical rather than classed (Borning, 1979; Borning, 1986; Ungar & Smith, 1987). That is, all interface objects are prototypes of a single abstract prototypical object and are therefore identical at creation. The user then builds on this abstract foundation, specifying the unique behavior of each object. There is no notion of inheritance or class-based specialization. This design choice reflects a decision to support the initial bottom-up design of interface objects from a set of fundamental parts as a construction kit (Fischer & Lemke, 1987), rather than as a top-down taxonomic specialization using inheritance. We feel that this approach lends itself to cognitive simplification. At the same time, we provide strong support for the two primary advantages of the class-based approach, abstraction and reuse. Abstraction is supported in that the user may define new "classes" by aggregating groups of objects, thereby defining a

[1] Note that in the commercial world, where standardization is important, it may actually be advantageous to have a firmly established library of interface objects. Thus, it may be desirable to sacrifice flexibility in favor of uniformity in these contexts.

new type of object; reuse is supported in that objects (simple or aggregate) may be duplicated and saved.

Thus, QUICK takes a more abstract view of interface construction: Instead of providing predefined classes, we provide a single fully general interface entity, allowing the user to define its specific attributes and behavior. The QUICK object may be viewed as a sort of shell, a basic framework which the user elaborates into a functioning interface component. This creative *elaboration*-based approach distinguishes QUICK from class-based systems which embody the *instantiation*-based perspective.

In sum, our approach represents a commitment to a prototypical (vs. a classical) view of objects, to *user*-defined object abstraction, and to reuse via a copy-edit approach.

Flexibility: An abstract interface language

With few exceptions, high-level toolkits do not allow the user to extend (by adding new classes) or modify the behavior of the predefined set of classes. Even when such taxonomic extension is possible (Goldberg and Robson, 1983), the user is forced to immediately move to a low-level language to accomplish the extensions. Thus, flexibility is highly constrained and extension is difficult or impossible. As noted above, the reasons for these limitations are rooted in the instantiation-based philosophy of these construction kits. In order to support the elaboration-based perspective taken in QUICK, which essentially views all interface construction as extension, it is necessary to provide an abstract interface specification language with which to accomplish the elaboration. That is, it would be of little use to provide the designer with abstract, unfinished interface objects with the expectation that they be elaborated using a low-level interface language. Thus, we have designed an abstract language, tightly focused on interface functionality, to allow the user to specify the behavior of individual interface entities.

To summarize, low-level languages offer a fully general interface language but provide no abstraction to control complexity. On the other hand, high-level languages are very abstract and structured, but provide little or no expressive freedom. In QUICK, we choose the middle ground, providing a moderately abstract prototypical interface object to structure the interface construction task, combined with an abstract interface language to maintain flexibility.

3.0 The QUICK Language

In this section, we will describe in (much) more detail the QUICK language. However, we begin with an example of an interface built with QUICK to serve as a firm foundation for our discussion and to illustrate the highly interactive direct manipulation interfaces QUICK is designed to produce.

3.1 An Example: The CardioLab

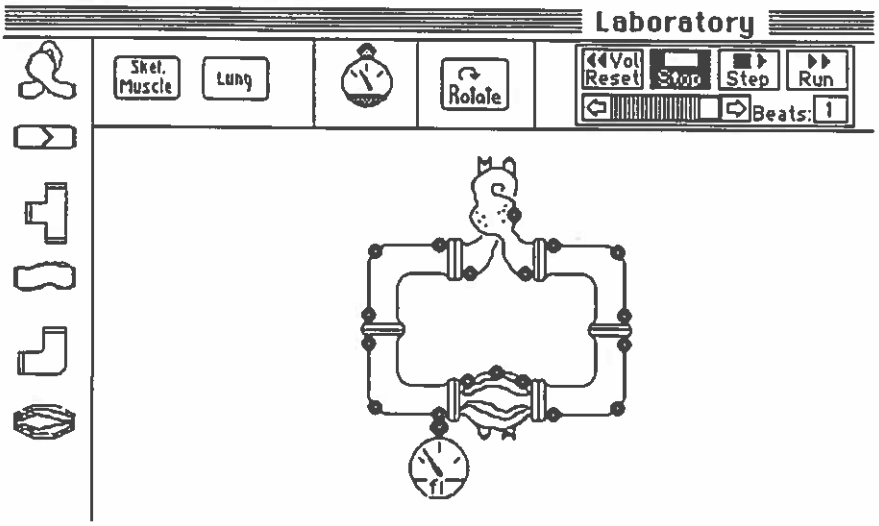
The following example illustrates the power and simplicity of the QUICK system and will serve as a descriptive vehicle throughout the paper. The interface shown below is a prototype interface for a cardiovascular toolkit, which we are developing in another project, to be used[2] in biology classrooms to teach various cardiovascular concepts, ranging from

[2] The CardioLab application has in fact been completed and has undergone extensive protocol-based evaluation. The application is now being used in actual biology lab classes and we are engaged in a final evaluation.

basic hydraulics to oxygen transport to comparative physiology. The interaction depicted in the following example reflects the relatively guided pedagogical style adopted for users of the application.

In a typical introductory session with CardioLab, the application user (student) is asked to construct a cardiovascular topology depicted in a diagram in the lab manual, which is accomplished by selecting icons from a "palette" of cardiovascular components, instantiating them in the workspace, and arranging them to form the desired topology. In particular, we specify for this example that a palette icon should become highlighted when it is single-clicked presenting a text definition of itself and un-highlighting a previously selected icon. Double-clicking the icon should cause a full graphic image of the component to appear in the workspace so that it can be dragged to position by the user. When two components are arranged so that they touch, both components should flash once to indicate that a connection has been made. In this manner, users construct cardiovascular systems with attached gauges to record behaviors of simulation variables. Finally, a control panel of buttons should be provided which, when clicked, should cause the simulation to commence, stop, reset, etc. During simulation runs, graphically animated arrows should indicate the direction of flow. Note that, since this is only a prototype of the interface, we do not expect the simulation to actually run and produce values -- all we require is that the presentational and user interaction behavior of the application, including object selection, dragging, flashing, highlighting and some forms of animation, be intact.

The following figure shows a view of the running prototypical interface, which was created in approximately one half hour (not counting time to draw the images).



3.2 QUICK Language: Semantics

We have seen that QUICK provides a very simple ontology based on a single abstract interface entity. This prototype interface object, through the mechanism of abstraction, defines a number of powerful but general behaviors. Designers may elaborate any number of such objects, using the compact and abstract QUICK language to specify the behavior of each. The interaction of all such objects defines the interface. It is important to note that from the point of view of QUICK, no difference exists between traditional but very general interface objects like menus and buttons, and custom interface objects like animated objects within a window.

Abstractly, the QUICK prototype object consists of four components: graphic representation, functionality, spatiality, and attributes.

Graphic Representation. In QUICK, the notion of an object's representation on the screen is disjoint from the object itself. That is, the object is viewed as a logical entity that can be represented in a number of ways. Two representations are predefined: the *image* of the object is an arbitrary picture created by the user in a drawing program; and the *text* of an object is an arbitrary string (initially the name of the object). Not only may the user switch between representations of the object, but both the image and the text of an object may be changed at any time. Indeed, one way to display values in QUICK is to create an object, ask it to display its text representation, and then simply change that text at run time.

Functionality. The issue of how to specify interface functionality is central in the design of an interface. Each object has associated with it a set of *actions*, specified in the QUICK user interface language, that define how it will respond to various events. Four built-in events are *1-click-event, 2-click-event, landing-events,* and *dragging-event* which specify what should occur when that object is single-clicked, double-clicked, dropped and dragged, respectively. For instance, in the CardioLab interface, the one-click-action of each workspace object causes that object to be highlighted and all others to be unhighlighted. In addition, the user may define new actions for any object along with the conditions under which they should be activated.

Spatiality. Our commitment to direct manipulation and the object oriented viewpoint has forced us to develop a clear framework for expressing spatial orientation and relations in the QUICK system.

Each QUICK object incorporates an explicit representation of its position in the interface, recording both its cartesian position the workspace and its *plane*, allowing the system to provide a simulated three-dimensional environment in which the designer can manipulate the relative "depth" of objects in the workspace. The explicit representation of position provides the designer with a means for expressing spatial relations between objects in the interface as well: it is possible to programmatically test spatial relationships[3] (e.g. in, above, left of, etc.) between objects. This feature has proved invaluable in designing the direct manipulation interfaces that are the target of our work. In the CardioLab example, for instance, the ability to test whether an object is *in* another object is crucial.

Attributes. Just as the functionality component defines an object's behavior, the attributes component defines its static characteristics. Built-in attributes specify whether the object is visible, draggable, highlighted, and a number of user-level characteristics. For instance, in the lifeboat example, the lifeboat's "draggable" attribute is set to false, making it a stationary component of the interface, while all other objects have the "draggable" attribute set to true, allowing them to be moved about. Naturally, the user may define new attributes (of any data type) and test and retrieve all object attributes programmatically.

3.3 Aggregation: Mechanism for partonomic abstraction

---

[3] In general, spatial relations (e.g. in, above, left of, etc.) have proven very difficult to define concretely, due largely to a strong dependence on the context and nature of the objects being related. We base our algorithms on past research in this area by Douglas et. al (1987).

This feature supports object abstraction. Though we do not support the class based view of interface programming, we do feel that a powerful abstraction mechanism is desirable. Thus, instead of providing built-in abstractions (i.e. classes), we provide a mechanism for allowing the user to define abstractions. Using **Group**, the user can aggregate any number of individual objects into a more abstract compound object. For instance, the user might design a numeric keyboard in which the numbers, when clicked, flash and announce their name. During programming, each of the keys would be represented by a QUICK object. However, when complete, the entire set of keys can be aggregated into a new "keyboard" object. Not only does this promote abstraction in the current interface, it creates a new resource for later reuse: the entire aggregate can be saved and later reloaded as a component of other interfaces. For instance, one might use the same keypad for a calculator, a telephone answering machine, and a day&night teller machine.

3.4 Animation and Sound

An important aspect of QUICK, closely related to spatiality, is the incorporation of animation. Each object may have associated with it a *path*, which the user records in a "by-rehearsal" style (Gould & Finzer, 1981), by asking the system to watch (record) and then dragging the object through the desired motions. The object can be asked to run its animation, moving itself along the predefined path.

We have invested considerable effort in exploring how best to support the user in specifying animation in the interface, an area that has received no attention up until now. For instance, it should be possible to specify that a given motion is to be repetitive, cyclic, relative to the current position or to some fixed point, and so on. QUICK accomplishes these goals by allowing the user to edit the recorded path, to specify the number of repetitions, and to select animation characteristics along three orthogonal dimensions: *mode, continuity,* and *orientation.* Briefly, mode is either *Jump* or *Drag* and specifies whether the object moves between adjacent path coordinates in a single leap or whether it moves more slowly along an interpolated line between them. The continuity may be *1-way, cyclic,* and *2-way* specifying what happens when the object reaches the end of its path with one or more repetitions left to go. In the first case, it runs its course[4] again, starting where it left off, in the second it first jumps back to where the animation originally started, and in the third, it simply reverses its path. Finally, the orientation may be either *fixed* or *relative* specifying whether animation should start from the object's current position (at the time it is asked to animate itself) or from some user-specified fixed point every time.

Taken together, these features provide a powerful and flexible means off specifying how an object should behave when asked to animate itself. Note that it is possible to ask several objects to simultaneously animate themselves. While this is very nice in that it allows multiple interface components to be in motion at once, it raises a new challenge: How can we allow the user to specify timing constraints to synchronize the activity. For example, the user may want to express "animate objects Foo and Fum, but fix it so that they reach the intersection of their paths *at the same time*". Addressing this issue remains a part of our agenda for the future.

Another unusual feature with extensive support in QUICK is sound. Most interface construction tools, if they support the generation of sound at all, simply allow the user to specify a file containing digitized sound to be played back at some point. QUICK takes a much more sophisticated approach by defining the concept of a *phrasal lexicon* which the designer can load with snippets of sound from various files. The designer may then "paste

---

[4] A path is represented by an array of offsets. Hence, movement can easily be initiated from any point.

together" various announcements using phrases in the lexicon. For instance, suppose the lexicon had been loaded with the following phrases: "The circle", "The square", "is", "above", "not", and "below". Then, by selecting different sequences of phrases, the designer could produce announcements like "The circle is above the square" and "The square is not below the circle". Lexicons may be saved, allowing related phrases to be grouped and loaded in together. Finally, lexicons may be linked to particular interfaces, causing them to be loaded when the interface is loaded.

Thus, we have provided sophisticated means of producing sound and animation in the interfaces constructed in QUICK. We see this as a natural consequence of our efforts to create a more flexible interface construction environment.

3.5 The Passive Background

One other QUICK components deserves brief mention. The *background* is simply a bitmapped image (as created by any of the popular painting programs) which serves as the passive backdrop for all of the action in the interface. This approach is much more efficient than defining the various lines, boxes, and other passive background elements in the interface as individual QUICK objects.

3.6 QUICK Language: Syntax

The QUICK language is extremely simple, compared to other approaches like data flow diagrams (Serius, 1989), forcing the user to program in the underlying language (Goldberg and Robson, 1983; Gould and Finzer, 1984), and even other high level interface languages (Apple, 1987). Thus, the QUICK language is a simple linearly organized object-oriented declarative language consisting of a small number of high-level commands. To further simplify QUICK, the language is embodied in a structure editor which guarantees that user programs will be free of syntax errors. The price we pay for this simple controlled environment is a loss of expressive power. However, it is not our goal to allow the user to express everything, but rather to discover and provide a minimal set of primitives that allow our unsophisticated user population to design a wide variety of interfaces without resorting to low-level programming.

The following figure outlines the QUICK language.

```
Animate <list of objects>
Flash <object>
Highlight <object>
Unhighlight <object>
If <test> <then> <else>
For-each <list of objects> <code>
Set-attribute <object> <attribute> <value>
Say <list of sounds>
Move <object> <location>
Trigger-action <object> <action>
```

Brief descriptions of the most important commands follow:

Animate allows the user to programmatically initiate animation. The argument is a list of objects which all simultaneously trace their user-defined animation paths.

IF is perhaps the most powerful command in the QUICK language, allowing the user to incorporate conditional branching into an otherwise linear language. Much of the power of the QUICK conditional stems from the variety of options available to fill the **<test>** argument. Among other things, the user may construct arbitrarily complex boolean statements by nesting AND, OR, and NOT constructs, test boolean attributes of any object, and test spatial relations between objects. The **<then>** and **<else>** arguments contain arbitrary code including, potentially, further IF statements.

For-each is a simplified looping construct. Each object in the list is, in turn, bound to the **self** variable after which the code in the **<code>** argument is executed. The user may select an *all-things* option to fill the **<list of objects>** argument, causing the loop to iterate over all interface objects extant at the time the statement is encountered.

Set-attribute allows the user to programmatically adjust any object's attributes. Note that the **<value>** slot may be filled by a constant or a computed expression.

Move causes the named object to be moved to the specified location. By default, the movement is along an interpolated path between the current position and the specified position. However, the user may change the speed and nature of the movement, just as with animation.

Trigger-action allows the user to programmatically trigger any actions (built-in or user-defined) listed for the named object. In other words, this command provides an alternative means (other than interface events) of triggering actions defined for an object. This has proved especially useful when used in conjunction with user-defined actions.

Trigger-action, in conjunction with the ability to create new user-defined actions for an object, fulfills our commitment to bring the power of object oriented programming to the novice interface designer. In object oriented terminology, the actions associated with an object can be viewed as *methods* while the Trigger-action command implements a *message passing* mechanism to activate those methods. As previously noted, we extend this paradigm with a unique twist: actions may also be activated by interface events.

4.0 Programming Environment

In the previous section, we argued that the interface language defines the conceptual framework of the interface programming task and determines its overall complexity. However, the interface programming environment plays an important supporting role in that it defines the *presentation* of the construction kit itself. Thus, the design environment in no way affects the expressive power of the system, but may help to streamline interface programming. Today, most interface construction kits rely upon various techniques that are generally categorized under *direct manipulation* systems. Direct manipulation was defined by Shneiderman (1983) to include three attributes: a) continuous representation of the object of interest; b) physical actions or labeled button presses instead of complex symbolic (typed) commands; and c) rapid incremental reversible operations whose impact on the object of interest is immediately visible.

Direct Manipulation has proven to be particularly useful in recent years. The reason for this success is that this feature is much more than simply a *pragmatic* environmental aid like autosaving or multiple windows. It is a *conceptual* aid, a "visual macro" that establishes a higher level of abstraction to move interface programming from the symbolic programming level towards a more direct visual model. In other words, direct manipulation provides a WYSIWYG environment for interface programming.

However, the above definition of direct manipulation is still too broad to clearly describe the differences between most user interface construction kits. We will attempt to develop some characteristics which can further refine our analysis. Firstly, most construction kits have some form of menu and/or dialog-box selection programming in which the object of interest is the text of the program itself. This characteristic we will call **weak direct manipulation**. Typically this form of direct manipulation saves the programmer from having to type in the symbolic text in the programming language. Structure editors are a more complex version of this type of programming in which the logic of the menu item contents changes depending on the syntax and semantics of the underlying language. Languages such as Smalltalk provide weak direct manipulation environments.

Strong direct manipulation programming environments add to the elements of weak ones by including elements of visual programming (Myers, 1986) and programming by demonstration (Myers, 1987). Visual programming refers to "any system that allows the user to specify a program in a two (or more) dimensional fashion." (Myers, 1986). Two dimensional here assumes graphical display and pointing devices. Visual programming in user interface construction kits often incorporates flow-chart style specifications of data-flow and general program control. Serius 89 is such an example and uses graphical data-flow specification. However, many user interface construction kits represent continuously the end-product of the programming process (i.e. the graphical interface itself) and allow the programmer to specify which object to program as a selection operation, or specify the location and animation paths of objects as a dragging operation rather than as symbolic coordinates. Prototyper and Hypercard illustrate visual programming techniques in positioning of button and icon placement in windows. Programming by demonstration involves having the programmer simulate the interactions of the user to indirectly specify the logic of user interaction. Peridot (Myers, 1987) and Programming by Rehearsal (Gould & Finzer, 1984) use programming by demonstration techniques.

Direct manipulation in interface construction kits contributes enormously to their popularity and the perception of increased productivity which they enjoy. However, there are several other features of programming environments which can also contribute to that goal. If a construction kit language is implemented as an interpreter rather than as a compiler, a rapid switch is possible between the programming tasks of specifying the program and testing it. This switch allows the programmer to test the program by simulating the behavior of the user. In addition, this switch also supports the rapid prototyping method of design. Because there are no known analytic methods which can completely predict the behavior of the user at the interface, designers frequently resort to rapid prototyping. Rapid prototyping allows the programmer to quickly try a variety of design options by testing them with users.

In summary, the interface programming language and the programming environment are orthogonal. That is, the environment establishes the presentation of the user interface construction kit itself, while the language the underlying conceptual model. In the next section we will discuss the QUICK system programming environment in light of the concepts we have developed here of weak and strong direct manipulation, since it incorporates elements of both.

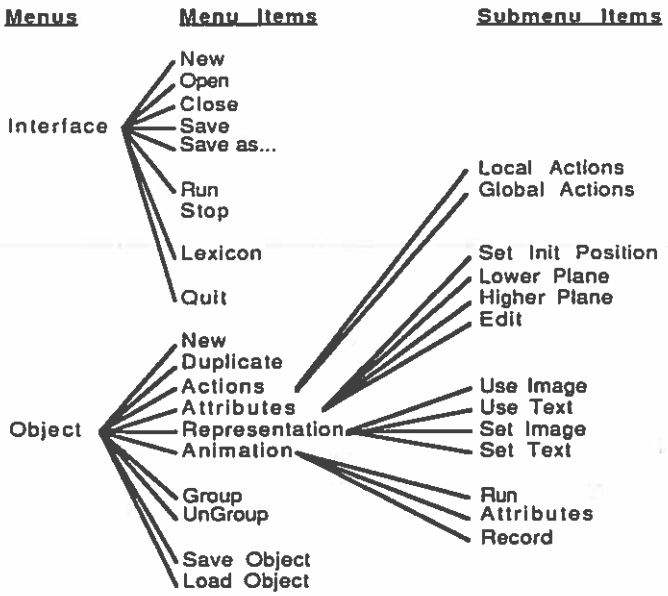
5.0 The QUICK Environment

5.1 The General Environment

The Quick programming environment supports both weak and strong direct manipulation. The objects of the user interface that the programmer is constructing are continuously visible at all times. As aspects of weak direct manipulation, QUICK provides menu control of its environment as well as a structure editor for using the QUICK programming language. As aspects of strong direct manipulation, objects to be programmed are selected by the mouse thus setting up internal program reference, objects are moved into position in the window thus indirectly specifying their coordinate locations, and animation paths are created by the programmer dragging the object rather than by symbolic specification of coordinates.

5.2 Control

Control of the QUICK programming environment is primarily accomplished via a hierarchically structure menu system while the elaboration of object behavior is specified with a structure editor. The structure of the menus is shown is the following figure.



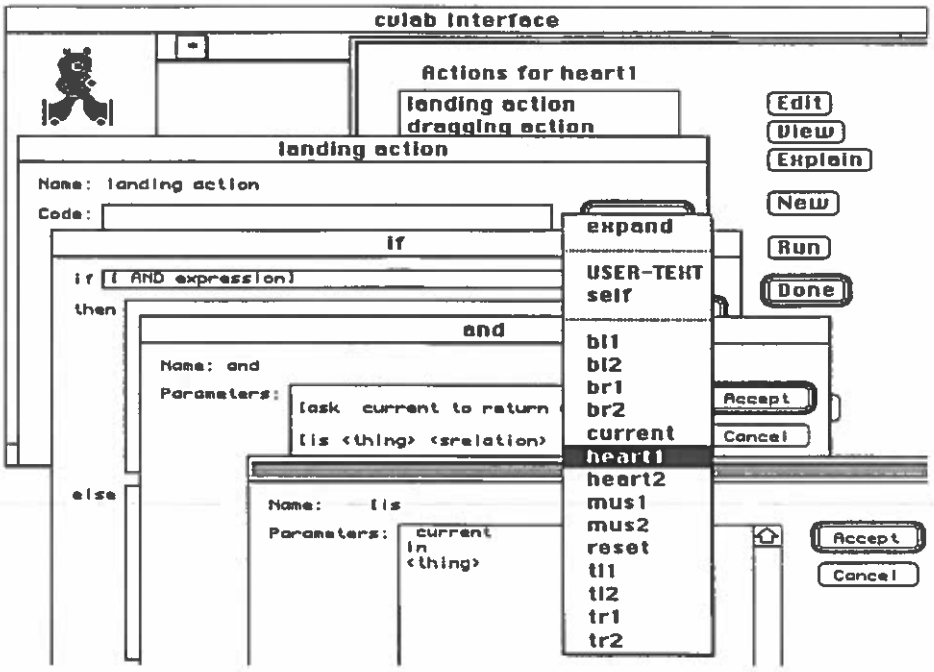
In general, the items in the **Interface** menu are concerned with the QUICK construction kit as a whole, while the items in the **Object** menu control operations on individual objects or groups of objects. Though detailed discussion is beyond the scope of this paper, several items warrant emphasis:

Run/Stop. These items control the *mode* of the system and implement the rapid prototyping feature of QUICK. At any time during programming, the user may select **Run** to instantly move the interface to run mode. In this mode, the interface behaves just as the final product will. Selecting **Stop** causes the interface to move back into programming mode. In this way, the user can quickly make changes, evaluate them, and make further modifications. Thus, QUICK supports a very tight modify-test iteration, a key feature of the rapid prototyping paradigm.

Duplicate. This important feature implements reuse in QUICK. The user can completely program a given object and then use the copy-edit approach to create a similar object.

5.3 The Structure Editor

Despite the simple nature of the QUICK language, programming would still be error-prone for novice users in a free form environment. Thus, we provide a hierarchically structure editor, which, at each stage, presents the user with only *legal* options to choose from. In fact, the only time the user is actually asked to type anything is when an object or file needs to be named.



The above figure gives the flavor of the structure editor. In the figure, we are in the process of programming the *landing-action* of the workspace Heart object of the CardioLab application. Recall that it must test whether it is connected to another workspace object, and to flash itself and the other object if so. The stacking of the dialogs illustrates the hierarchical nature of the editor: The user selects the *landing-action* to edit, is presented with a code dialog, selects **If** from a popup menu (no longer visible in the figure), and is presented with a specialized dialog for **If**. To fill the If's boolean test, the user has selected an AND expression, with the purpose of first testing to see if an object is a workspace object, and then testing to see that the Heart is IN that object. At the moment of this snapshot, the designer is completing the second of these conjunctive tests and has just depressed the mouse to display a popup menu of available objects to fill the <thing> slot of the spatial relation test and has selected the object Heart1.

Thus, the structure editor combines the *fill-in-the-blanks* and *multiple-choice* paradigms to create a highly constrained programming environment.

5.4 The CardioLab Revisited

In this section, we take a brief look at how a QUICK interface is actually constructed by showing how the CardioLab interface described earlier was implemented. For the sake of brevity, we will show only how the **heart** in the workspace is created and programmed. All other workspace objects are very similar, and were, in fact, created by applying the copy-edit approach to the **heart** object. The **heart** object is created and programmed in three easy steps:

1. **Create the object.** We simply select New from the object menu. The user is prompted for a name and asked to select an image for the new object.

2. **Set the attributes.** We want the **heart** to be draggable at run time so that the user can arrange it in the workspace. Since the default value is non-draggable, we select the heart by clicking on it, and then select **Edit** from the submenu of the **Attributes** item in the **Object** menu. This invokes an attribute editor for the attributes of **heart**, allowing us to toggle the value of the **draggable?** attribute. At this time, we also add a new boolean attribute, **w-obj?**, initially set to TRUE, which identifies the new object as a workspace object, differentiating it from a palette object and other objects we may create. We will see how this is useful below.

3. **Program the new object.** According to the specifications above, we must provide programs for two of the built-in interface events:

   1-click-action. The **heart** should simply highlight itself and unhighlight all other workspace objects.

   Landing-action. When the object is dropped, it should determine whether or not it is touching another workspace object and, if so, both objects should flash once and the system should produce an audible "click" to indicate a connection.

The code for the **heart** object is shown in the following figure:

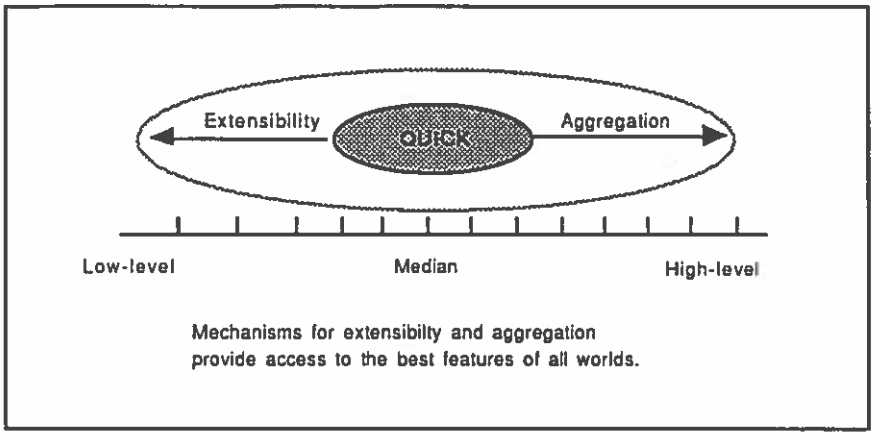
```
Landing action:
   FOR-EACH  All-Things DO   ;  For each object, each bound to CURRENT in turn
      IF [AND
           [the value of CURRENT's w-obj? attribute is TRUE]  ; check if it's a workspace object AND
           [A-thing is IN Heart]]                             ; check if it's IN the heart
      THEN
        FLASH  CURRENT
        FLASH  Heart
        SAY ["Click-click"]  ; Give audio feedback to indicate connection.

One-click-action:
   FOR-EACH  All-Things DO          ; Iterate over all objects, CURRENT bound to each in turn
     IF [the value of CURRENT's w-obj? attribute is TRUE]  ; If it's a workspace object
     THEN
        UNHIGHLIGHT CURRENT
   HIGHLIGHT Heart
```

6.0 Discussion

6.1 Stretching limitations: How to have your cake and eat it

As shown in section 2, the extremes of the interface language spectrum each have unique advantages and can be supported or not by the programming environment. The low-level maximizes expressive power while the high-level provides maximum abstraction. By settling on a middle ground, at the median level, we have minimized *overall* interface programming complexity --- but this has obviously and necessarily resulted in a compromise in terms of both expressive power and abstraction. Is there a way that a toolkit can still maintain access to these features? We believe so and have worked to incorporate features which allow the designer to "stretch" the functionality of the QUICK system towards the extremes, as shown in the figure below.



Mechanisms for extensibilty and aggregation provide access to the best features of all worlds.

By providing mechanisms for *aggregation*, we support the assemblage of primitive interface entities into more abstract ones. Strong support for *extensibility* allows the user to easily incorporate fundamental level code in an evolving design. These aspects are discussed in more detail in the following sections.

6.2 Creating high level interface entities through aggregation

A major theme of this paper has been that, by settling on a median level of abstraction, somewhere between low-level and the high-level interface languages, an interface construction kit can provide unusual flexibility without introducing overwhelming complexity. While this feature is especially advantageous for non-programmers and designers of non-standard interfaces, it is important to point out that one can easily model standard interface entities like windows, menus, and buttons in a median level system as well.

In this section, we present a short example that illustrates how QUICK can be used to create a simple "pull-down" menu. The approach is as follows: view the menu in terms of its component pieces; create each piece as a separate QUICK object, specifying the appropriate behavior; and juxtapose the components via direct manipulation. Importantly, the set of component pieces may then be aggregated (partonomic abstraction) into a new user-defined interface entity that can be stored and reused in later designs.

We decompose a pull-down menu into the following components:

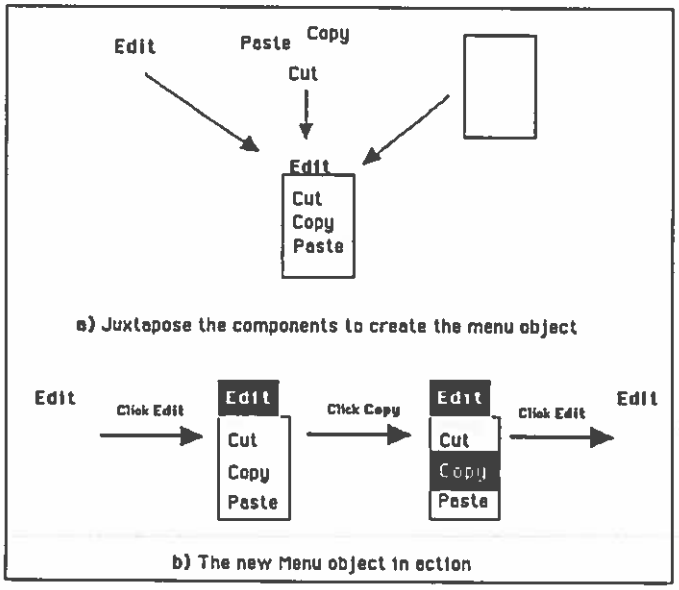
A menu title:
Representation: The descriptive title that appears in the menubar.
Behavior: When single-clicked, highlight itself and cause the menu background and items to become visible.

A menu background:
Representation: A blank oblong box large enough to contain all menu items.
Behavior: None.

Menu items:
Representation: Text items representing the choices in the menu.
Behavior: Single-clicking causes item text to flash and then initiates the action define for that item.

The figure below shows how primitive menu components are composed to create an interface entity at the composite tool level. Now suppose the designer aggregates the five component objects into a new compound object under the name **menu-object**. If at a later

point (in the same design or during new one) the designer requires another menu, the stored **menu-object** can be loaded in and trivially modified to fit the new task. Modification would primarily involve changing the text strings of the menu title and items and the assignment of appropriate actions to each item.



In sum, the less-structure approach to interface design taken in QUICK in no way impedes designers of "standard" interfaces. By way of partonomic abstraction, designers may compose an unlimited variety of higher level interface entities, including widely used entities like windows, menus, and buttons. In this way, QUICK provides all of the time-saving power associated the composition-based approach to interface design, in which predefined interface components are selected and composed, while at the same time maintaining flexibility and the ability to design unique interfaces. Thus, QUICK is well-suited to the needs of both designers interested in creating uniform, standardized interfaces as well as designers wanting to explore new concepts in interface design.

6.3 Extensibility for greater expressive power

For some programs which are interface intensive, with very little underlying computation to be done, the interface generated by QUICK *is* the finished application. That is, there is no further programming to be done. However, in the case of more complex software, like the CardioLab example presented earlier, which may involve intricate data structures and elaborate computations, there are **inherently** cases where desired behavior cannot be specified by the toolkit. To see this, recall that the interface design toolkit reduces interface programming complexity by introducing a set of abstractions, specific to the interface programming domain. These abstractions structure *and constrain* the space of possible programs to varying degrees, depending on the specificity of the abstraction. For instance, the windows and menus available in high-level interface design kits are highly specific abstractions, and therefore rigidly constrain the nature of interfaces that can be produced. QUICK, which provides a median level of abstraction, is accordingly less rigid and allows a wider range of interfaces to be specified. But irrespective of the level of abstraction chosen, the sophisticated design kit user will, at some point, need to implement functionality which simply is not supported by the abstractions embodied in the design toolkit. At this point, the designer must resort to a less abstract specification language,

most commonly the low-level language on which the toolkit is based. Thus, a toolkit which supports design at a lower level of abstraction (but one above the low-level) may postpone, but not avoid, the need to resort to low-level programming.

In these cases, the interface construction kit must become a tool to streamline the mundane aspects of interface construction, freeing human programmers to focus on unusual interface effects and computational aspects beyond the scope of the interface toolkit. The QUICK system strives to support this goal through a variety of features:

- QUICK is a generates fully functional Lisp code which can be arbitrarily extended and modified by the designer.
- Unlike many other toolkits, code the has been modified by the user may still be loaded by the QUICK system allowing further changes to be made.
- The QUICK language provides a mechanism for referencing values produced by arbitrary Lisp objects. Requests for such values are simply incorporated into the generated code and are resolved at run time.
- The QUICK language is fully extensible. The user may create a library of personalized interface functions to extend the primitives provided by the QUICK language.

The purpose of this section has been to emphasize that, while the simplicity of QUICK makes it an especially promising tool for unsophisticated designers, this feature in no way compromises its power as an interface design tool. By aggregating simple interface entities, it is possible to model more complex entities like buttons and menus. Though the median level of abstraction adopted in QUICK allows a wide range of interfaces to be created without leaving the QUICK language, the system provides a variety of mechanisms to support low-level elaborations.

7.0 Designer's Evaluation of QUICK

We feel that formal evaluation of software has been greatly neglected in recent work and have a strong commitment to reversing this trend. Thus, we are currently engaged in two distinct evaluative efforts.

Our first study works to evaluate the utility of QUICK as an interface design tool for programmers. We plan to gauge utility by combining several metrics: coding time, coding correctness, and attitude. Importantly, we are interested in gaining insight into the contribution of the direct manipulation and object-oriented aspects of QUICK to its overall success as an interface toolkit. For instance, we are curious about whether subjects will reuse objects and how they fare with the prototype model of copy and edit (as opposed to the standard object-oriented model of class hierarchy specialization). A study by Lange & Moher (1989) will give us classifications of coding strategies that pay particular attention to reuse.

The study involves ten pairs of subjects working in pairs. We have found this technique, known as *constructive interaction* (Miyake, 1986), to be much more productive than the well-known "think-aloud" approach (Newell and Simon, 1972) in providing insight into a subject's problem solving process. Each pair is shown a 30 minute videotape of how to use QUICK and then asked to implement three distinct interface specifications. While

apparently quite different, the three problems are designed to provide opportunities for reuse. The resulting videotapes are then analyzed to give the above performance measures.

Preliminary results indicate that, even with minimal training, programmers are able to produce simple interfaces using QUICK. However, several disturbing trends are beginning to emerge as well. First, though subjects have the opportunity for reuse, both within (via copy-edit) and between interface design efforts, they generally fail to do so. We feel that part of this is due to a failure to detect *functional* similarity between objects in the same as well as different designs. This is an area where a more interesting study could be done based on existing cognitive psychology research in analogy. We also note that a major problem with reuse is that frequently designers fail to predict what might be useful in a future design and consequently don't save objects either as individuals or aggregate objects.

Our second study is more general and addresses the question of how useful QUICK is when compared a) to low-level interface programming and b) to other interface design toolkits. A problem in addressing the first objective is that the only expert QUICK programmers available are the four designers, only two of whom are still at the University. Therefore, we are able to report a very preliminary result. We asked a QUICK expert to use the system to prototype a direct manipulation cardiovascular simulation interface developed on another project. The programmer of the original interface, working in Allegro Common Lisp, took approximately 500 hours (six months) to complete. Our QUICK expert produced a prototype in approximately six hours. We believe that much of this improvement is due to the direct manipulation techniques and the high level object-oriented QUICK language. With object-oriented programming Schmucker (1986) estimates that development time can be reduced by a factor of four or five.

8.0 Conclusions

The concept of interface construction kits has become become firmly established and well-accepted in recent years. Through both their popularity and the perceptions of their users, they seem to offer solutions to the programming complexity created by introducing user interface operations into traditional programming languages. We have argued in this paper that an understanding of the productivity gain afforded by construction kits must be shaped by separating programming language from programming environment.

We have argued that abstraction and the creation of high-level languages, particularly object-oriented ones, reduces programming complexity. We have also noted that when languages strive for abstraction they also strive for generality, thus losing programming power and expressiveness. Our own research, the QUICK user interface construction kit extends this research by exploring the middle ground in language abstraction. Our use of the prototype model of object-oriented languages rather than the class-based system is a relatively unique contribution.

We have also described characteristics of user interface construction kit programming environments making a distinction between weak direct manipulation environments and strong ones. Weak environments essentially support the program text as the object of interest and allow text object, menu and dialog box selection of programming language components. Structure editors are an elaborate version of this. QUICK is definitely a weak direct manipulation environment. Strong direct manipulation environments feature elements of visual programming and programming by demonstration. QUICK is also a strong direct manipulation environment, allowing the programmer to continuously display

the product user interface and directly specify programming constructs by positioning and movement of those objects. We also noted that some programming environments allowed rapid switches between programming and testing. This supports the concept of rapid prototyping in design. QUICK is one such system.

The prototype object oriented language paradigm, weak and strong direct manipulation programming environments, and rapid prototyping support combine in a highly simplified system designed to reduce the implementation complexity of user interfaces. By avoiding the limitations associated with a fixed, predefined library of interface entities often found in other kits, and, instead, allowing *the user* to define new abstractions, we have created a construction kit that is extremely flexible. Not only can we emulate the entities provided as primitives (e.g. buttons, pull-down menus, etc.) in other construction kits, we are able to incorporate unusual features like sound and animation. One side-effect of these features is the creation of a user interface construction kit that is useful enough for experienced programmers and novices alike while remaining easy for them to learn.

Our goal has been to identify a set of representational and functional abstractions that are conceptually simple but powerful enough to allow construction of a wide variety of user interfaces in a relatively short period of time. Our evaluation studies to date support that claim.

The QUICK system is an interpreter implemented on the Macintosh II computer in Allegro Common Lisp using Allegro's Object Lisp system. It generates Lisp code which can be supplemented by additional LISP programming if necessary. Both graphic images and digitized sounds are created by other pieces of software and easily imported into QUICK. The source code is roughly 700K in length.

9.0 Acknowledgements

This research has been funded in part by a grant from the U.S. Department of Education (FIPSE) and by a contract from US WEST Advanced Technologies Lab. We would also like to acknowledge the early encouragement of Laura Gould creator of *Programming by Rehearsal* and the work of Gary Smithrud, who programmed the first versions of QUICK.

10.0 References

Apple. *Hypercard User's Guide*. Apple Computer, Inc., 1987.

Barth, P.S. "An Object-Oriented Approach to Graphical Interfaces". *ACM Transactions on Graphics, Volume 5* (2):142-172, April 1986.

Borning, A. *ThingLab - A Constraint-Oriented Simulation Laboratory*. Unpublished doctoral dissertation, Stanford University, March 1979.

Borning, A. "Classes versus Prototypes in Object-Oriented Languages". In *Proceedings of the ACM / IEEE Fall Joint Computer Conference*, Dallas, TX, November 1986.

Cardelli, L. *Building User Interfaces by Direct Manipulation.* DEC Systems Research Center Technical Report #22, October 2, 1987.

Douglas, S.A., Novick, D.G. and Tomlin, R.S. "Consistency and Variation in Spatial Reference." In *Proceedings of the Ninth Annual Conference of the Cognitive Science Society*, 1987.

Fischer, G. and Lemke, A. C. "Construction Kits and Design Environments: Steps Toward Human Problem-Domain Communication". *Human-Computer Interaction*, 3: 179-222, 1988.

Fischer, G. and Lemke, A.C. "Knowledge-Based Design Environments for User Interface Design". Submitted to the 11th International Conference on Software Engineering.

Goldberg, A. and Robson, D. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley Publishing Company, Reading, MA., 1983.

Gould, L. and W. Finzer, *Programming by Rehearsal.* Technical Report #SCL-84-1, Xerox PARC, Palo Alto, Calif., 1984.

Henderson, D.A. "The Trillium User Interface Design Environment." *Proceedings of the Human Factors in Computing: CHI'86*, Boston, MA, April 1986, 221-227.

Lange, B. and Moher, T. "Some Strategies for Reuse in an Object-Oriented Programming Environment.." *Proceedings of the Human Factors in Computing: CHI'89*, Boston, MA, April 1989, 221-227.

Miyake, N. "Constructive Interaction and the Interative Process of Understanding" *Cognitive Science*, 10: 151-177, 1986.

Myers, B. "Visual Programming, Programming by Example, and Program Visualization: A Taxonomy," *Proceedings of the Human Factors in Computing Systems Conference: CHI '86*, Boston, MA, April 1986.

Myers, B. "Creating Dynamic Interaction Techniques by Demonstration," *Proceedings of the Human Factors in Computing Systems and Graphic Interface: CHI + GI 1987*, Toronto, Canada, April 1987.

Newell, A. and Simon, H. *Human Problem Solving*. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1972.

Schmucker, K. "MacApp: An Application Framework." *BYTE*, 11, 8, 1986.

Serius89. *Serius89 Documentation Release 1.1*. Serius Corportation, 1989.

Shaw, M. "An Input-Output Model for Interactive Systems" *Proceedings of the Human Factors in Computing: CHI'86*, Boston, MA, April 1986, 261-273.

Shneiderman, B. Direct Manipulation: A Step Beyond Programming Languages, IEEE Computer, Vol. 16, No. 8, 57-69, 1983.

SmethersBarnes. *Prototyper User's Manual*, SmethersBarnes Publishing Division, Portland, OR., 1989.

Smith, R.G. On the Development of Commercial Expert Systems, *AI Magazine*, Vol. 5, No. 3, Fall 1984, 61-73.

Ungar, D. and Smith, R.B. "SELF: The Power of Simplicity". In *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference.* Orlando, Florida, October 1987.