The Garnet User Interface Development Environment; A Proposal

Brad A. Myers

September 1988
CMU-CS-88-153

Abstract

The Garnet project aims to create a set of tools that will help user interface designers create, modify and maintain highly-interactive, graphical, direct manipulation user interfaces. These tools will form a "User Interface Development Environment" (UIDE), which is sometimes called a "User Interface Management System" (UIMS). Garnet takes a new approach to UIDEs by concentrating on a particular class of programs: those whose primary focus is creating and editing graphical objects. Garnet is composed of six major parts: an object-oriented graphics package, a constraint system, encapsulated input device handlers called "interactors," a user interface tool kit, user interface construction tools, and a "graphical editor shell" to help build editor-style programs. This document presents an overview of the approach that we propose to take for the Garnet project.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, Amendment 20, under contract F33615-87-C-1499, monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), Wright-Patterson AFB, Ohio 45433-6543. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

Table of Contents

1. Introduction 1
2. Importance 3
3. Goals 3
4. Related Work 4
5. Parts of Garnet 5
   5.1 Object-Oriented Graphics 7
   5.2 Constraints 9
   5.3 Interactors 11
   5.4 Construction Tools 15
   5.5 Tool kit 16
   5.6 Graphical Editor Shell 17
6. Implementation Issues 18
7. Validation 19
8. Conclusions 19
Acknowledgements 20
References 20

1. Introduction

Garnet is a new research project in the Computer Science Department at Carnegie Mellon University. Garnet stands for Generating an Amalgam of Real-time, Novel Editors and Toolkits, and is designed to help build graphical, highly-interactive, direct manipulation [Shneiderman 83] interfaces. The principal investigator is Dr. Brad Myers, and Drs. Roger Dannenberg and Dario Giuse are additional investigators. Other researchers on the project include Dr. Pedro Szekely, Dr. Brad Vander Zanden and Philippe Marchal. Lynn Baumeister and Jake Kolojejchick are research programmers.

The primary goal of Garnet is to provide an environment in which programs that make heavy use of the mouse to manipulate graphical objects on the screen can be easily created. These programs can be called "graphical editors." One important characteristic of graphical editors is that, although there are many aspects in common, each one has different application-specific properties. The various parts of Garnet are designed to provide the common aspects in a central place while making application-specific functionality easy to specify.

The Garnet project is implemented on top of the Mach operating system [Rashid 88] (which is a version of Unix) in CommonLisp [Steele 84] on top of the X window system [Scheifler 86] (and possibly also on NeWS [Sun 86], see section 6). It is composed of six sub-parts as shown in Figure 1:

- An *object-oriented graphics system* where graphical objects such as lines, rectangles, and text strings can display and erase themselves and handle refresh automatically.
- *Interactors* that encapsulate iteractive behaviors independent from the graphics.
- A *constraint system* that allows relationships among graphical objects to be specified easily and then maintained automatically even if the objects are modified.
- A *user interface toolkit* that contains a collection of interaction techniques, such as menus, scroll bars, buttons, sliders, etc.
- A *Graphical Editor Shell* that will help create graphical-editor-style programs. "Shell" here is used in the same way it is used in "expert system shell:" the outer layer (shell) of a program that supports the rest of the development.
- *Graphical Construction Tools* that will allow all pictorial parts of the user interface to be drawn graphically without programming.

This document presents an overview of the Garnet project. Since the research described is in progress, the final form for various parts may differ significantly from what is described here, but the overall goals and direction are expected to remain the same.

First, however, it is important to define a few terms. A *User Interface Development Environment* (UIDE) is a software program or set of integrated programs that helps a user interface designer create and manage the design and/or implementation of user interface software. UIDEs are sometimes called *User Interface Management Systems* (UIMSs). The term UIMS is not used here, however, since it often is used for systems that only address limited dialogue issues, and Garnet has a larger scope.

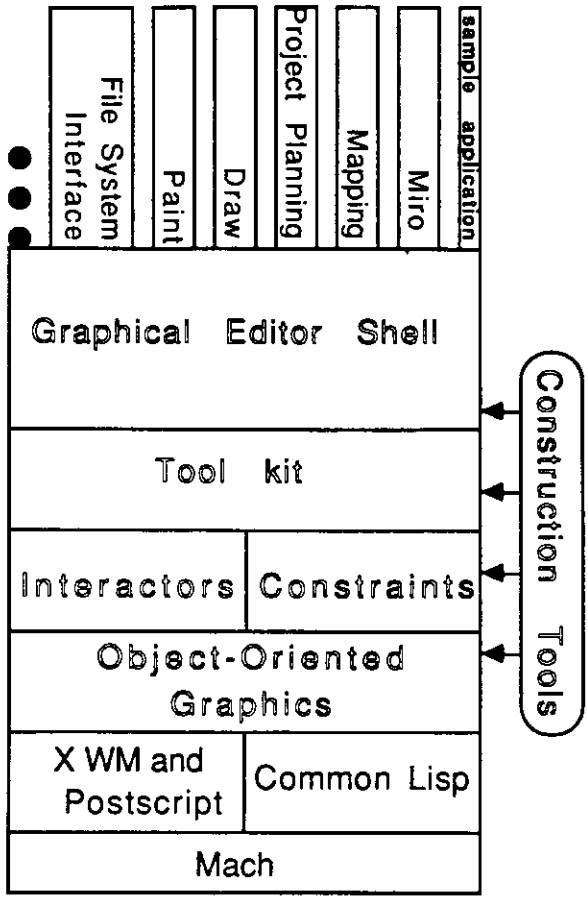


Figure 1.

An overview of the parts of the Garnet User Interface Development Environment. The names in outline font are parts of Garnet (see section 5). At the top are the applications built using Garnet, and on the bottom is the Mach operating system.

There are four different classes of people involved with any UIDE and it is important to have different names for them to avoid confusion. One person is the designer of the UIDE (e.g., me), who is called the *UIDE creator*. The next person is the designer of a user interface, and will use a UIDE. This person may be a programmer or a graphic artist, depending on the specification technique used by the UIDE, and will be called the *user interface designer* or just *designer* for short. Another person involved is the programmer that creates the application program which is connected to the user interface created by the user interface designer. This person is the *application programmer*. Finally, there is the person who actually uses the final product. This person is the *end user* or just *user*. Note that although this classification discusses each role as a different person, in fact, there may be many people in each role, or one person may perform multiple roles.

2. Importance

It is generally recognized that creating a good user interface for a system is a difficult task, and a large number of existing programs have very poor user interfaces: they are hard to learn, hard to remember, inefficient to use, have limited on-line help, etc. In addition, user interface software is often large, complex, and difficult to debug and modify [Myers 89, Olsen 87]. The user interface for an application is usually a significant fraction of the code. One study found that the user interface portion was between 29% and 88% [Sutton 78]. In artificial intelligence applications, for example, surveys report that 40% to 50% of the code and run-time memory are devoted to user interface aspects [Bobrow 86, Mittal 86].

The easy to use "direct manipulation" [Shneiderman 83] graphical-editor style interfaces that Garnet addresses are among the most difficult kinds to implement [Williams 83, Smith 82]. Some reasons for this are that they usually provide:

1. elaborate graphics,
2. multiple ways for giving the same command,
3. multiple asynchronous input devices (usually a keyboard and a locator or pointing device such as a mouse),
4. a "mode free" interface, where the user can give any command at virtually any time, and
5. rapid "semantic feedback." *Semantic feedback* is when determining the appropriate response to user actions requires specialized information about the objects in the program. An example is in the Apple Macintosh user interface, where icons highlight when another icon is dragged over them if they perform semantically meaningful operations on the icon being dragged.

User interface tools have been demonstrated to make creating user interface software significantly easier. For example, Apple's MacApp has been reported to reduce user interface software development time by a factor of four or five [Schmucker 86].

3. Goals

The primary goals of the Garnet project are:

- To create a small set of interactor objects that cover a wide range of user interface styles of interaction.
- To demonstrate that the use of constraints and interactors make the construction of user interfaces and user interface toolkits easier, more modular, and more modifiable.
- To demonstrate that it is possible to provide graphical, direct manipulation construction tools that allow significant parts of the user interface to be constructed without programming.
- To provide a Graphical Editor Shell that will help build graphical editor style programs. This involves identifying the common functionality and incorporating it into a useful program.

4. Related Work

A common name for software that builds user interfaces is a "User Interface Management System" or UIMS, and there are many examples of these ([Myers 89] is a survey). Unfortunately, most of these programs have serious problems and have not been widely used. Exceptions include Apollo's Dialogue [Schulert 85] and MacApp [Schmucker 86].

The primary influence on the Garnet project is my Peridot UIMS [Myers 86a, Myers 87, Myers 88]. It is a construction tool that allows toolkit items to be created without programming. Peridot, which stands for Programming by Example for Real-time Interface Design Obviating Typing, allows non-programmers to create many types of interaction techniques, including most kinds of menus, property sheets, light buttons, radio buttons, scroll bars, two-dimensional scroll boxes, percent-done progress indicators, graphical potentiometers, sliders, iconic and title line controls for windows, and many others. Thus, Peridot can create almost all of the Apple Macintosh interface, as well as many new interfaces, such as those which use multiple input devices concurrently. Peridot also created its own user interface. Like Garnet, Peridot uses constraints and interactors, but there is no programming interface to any Peridot features; it is a stand-alone program. Garnet, on the other hand, is designed so that its parts can be used independently by programmers, as well as by the Garnet construction tools.

Another influence on Garnet are interaction technique layout tools such as the two Macintosh programs: Prototyper from Smethers Barnes and the Exper User Interface Builder. Examples from research labs include Menulay [Buxton 83] and one from DEC SRC [Cardelli 87].[1] These programs allow the user interface designer to place pre-programmed menus, scroll bars, and buttons in windows, and then typically allow the designer to type in the name of a procedure that should be executed when the interaction technique is executed. These tools do not allow aspects of the interaction techniques themselves to be edited, however. The Garnet construction tools are designed to be as easy to use as these programs when the user only wants to assemble interaction techniques, but also to be more functional when new ones are desired.

Constraints have been used by many systems, starting from Sketchpad [Sutherland 63], and later Thinglab [Borning 79]. The GROW system [Barth 86] uses constraints as part of a user interface tool kit, and is therefore more relevant to Garnet. Another use of constraints is Vander Zanden's system, which uses "constraint grammars" that synthesize ideas from attribute grammars and constraint-based object models to produce a more powerful specification language for graphical interfaces. It also presents incremental algorithms for solving noncircular, multidirectional constraints, thus allowing many constraint grammar specifications to be automatically and efficiently implemented [Vander Zanden 88]. Garnet provides an easy-to-use syntax for specifying constraints, constraints on sets of objects, and the ability to define constraints in the abstract and apply them to specific objects later.

---

[1]Note that Cardelli uses the term "interactor" for this paper calls "interaction techniques" (menus, scroll bars, buttons, etc.). "Interactors" in Garnet are quite different; they are encapsulations of input device behavior devoid of any graphics (see section 5.3).

There have been many attempts to separate input devices from application programs. The approach used by all of the graphics standards (PHIGS, GKS, CGI, CORE, etc.) identifies 5 or 6 basic input types (locator, stroke, valuator, choice, pick and string for PHIGS [PHIGS 84]). Unfortunately, these are implemented inside the graphics package, so, for example, the programmer has no control over what kind of menu is used when a "choice" input is requested. In general, these input classes have proven unacceptable for implementing highly-interactive user interfaces [Meads 87].

A different approach was taken by Smalltalk [Goldberg 83] with the "model-view-controller" (MVC) paradigm. The idea here was to separate the code into three parts: the *model* which embodies the application semantics, the *view* which handles the output graphics that showed the model, and the *controller* which handles input and interaction. The goal is that it should be possible to modify any part, for example, to change the way an item looked on the screen, without affecting the other two. Unfortunately, programmers found that the code for the controller and view were tightly interlinked, so modern object-oriented systems have not used any separation, or, like the Andrew toolkit [Palay 88], have only used two parts: the view (which handles all input, output and interaction) and the model. Garnet attempts to achieve the separation goal of the MVC paradigm by using interactors to handle the interactive behavior (the "controller"), object-oriented graphics for the "view," and conventional Lisp code for the "model." Constraints are used to tie the parts together.

Another important influence on the Garnet project is Apple's MacApp application development environment [Schmucker 86]. MacApp provides an object-oriented framework that helps build programs with the standard Macintosh user interface. MacApp handles many of the details of the user interface, but leaves a number of hard problems to the application developer. In particular, while MacApp handles the menus and scroll bars *around* a window, it does not help much with mouse or keyboard events *inside* the window. For example, the application is told about mouse button presses and movement, and is required to deal with all issues of selection and moving objects itself. Garnet's Graphical Editor Shell will attempt to go further in this area by helping to deal with input events inside the application's window also.

5. Parts of Garnet

As shown in Figure 1, Garnet is built on top of Common Lisp and the X window manager. It is composed of 6 parts which are described in this section. We expect many different application programs to be built on top of Garnet, as explained in section 7.

The *object-oriented graphics component* of Garnet allows the higher layers of the software to be independent of the details of the particular window manager used. In particular, this layer supports "retained graphics," which means that the objects know where they are displayed in a window and are able to redisplay themselves automatically if the window is uncovered, and they can move and erase themselves.

A *constraint* is a relationship among graphical objects that is maintained even if one of the objects changes. Constraints are a natural way to express the kinds of relationships that are common in graphical user interfaces. For example, in an editor that supports boxes attached by arrows, the user interface designer can specify a constraint that the arrows must stay attached to the boxes. Then the boxes can be moved by the program or the mouse, and the lines will stay attached without any additional coding.

One goal of most user interface development environments and toolkits is to free the application from details of specific interactors and input device behaviors [Pfaff 85]. This has proven to be a very difficult goal, and most previous attempts to achieve this separation have failed. Garnet therefore is taking a new approach, and identifies a few low-level input device behaviors. These are encapsulated into objects called *interactors*. There are a small number of different types of interactors, and each one handles a different kind of input device interactive behavior. For example, there are interactor types for menu behavior (selecting one from a set of objects), moving behavior (moving an object with the mouse), and angular rotation behavior (for interacting with circular gauges, etc.). In each case, the output graphics is entirely independent of the behavior, which allows tremendous flexibility.

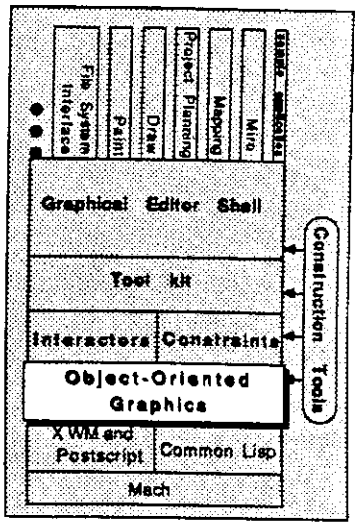
The next layer is a *toolkit* which will be built using the object-oriented graphics, constraints, and interactors. We believe that using constraints and interactors will make the toolkit items much easier to build and modify, and therefore demonstrate that this is a more appropriate way to build toolkits than the conventional straight object-oriented programming approach.

The top layer of the Garnet system will be the *Graphical Editor Shell*. This will be a highly customizable graphical editor that will make it easy to create specialized, application-specific editors. The goal of the Graphical Editor Shell will be to centralize all of the common elements of editors while allowing applications to specialize appropriate aspects easily.

All good user interface development environments come with a set of *construction tools*. These are programs that help user interface designers create parts of the user interface. They may be as simple as icon editors and font editors, or they may allow sophisticated layout and prototyping to be performed. The Garnet construction tools will attempt to be the most sophisticated yet created for graphical user interfaces. They will support layout of toolkit items and graphical decorations, as well as icon editing. An important additional capability will be the ability to modify the graphics and behavior of toolkit items. For example, it should be possible to define a new type of menu by editing the standard menu definition. This will allow the construction of custom "looks and feels" for user interfaces. In addition, the construction tools will allow application-specific graphical entities to be designed. The application will still control the creation and manipulation of the entities, but the construction tool will allow the graphics and some standard behaviors to be specified.

The following sections describe these parts of Garnet in more detail. Because the design of Garnet is still in progress, many of the details are speculative and are likely to change. There will probably be separate technical reports about all of the parts when their designs are more stable.

5.1 Object-Oriented Graphics



The object-oriented graphics layer is not intended to be especially novel or innovative; there have been many object-oriented graphics packages before. The important aspect of this part is to enable the other parts of the system to be easily built. This layer will also be usable independent of the other parts of Garnet, in case the constraints and other tools are not needed by some applications.

The object-oriented graphics will form the interface between Garnet and the window manager for graphics output (the interface for input is handled by the interactors described in section 5.3). Garnet is currently being implemented on top of the X window manager [Scheifler 86], version 11, using the standard CommonLisp X interface, called CLX. We plan to support output to Postscript either for Display Postscript or the NeWS window manager (see section 6). Therefore, we hope that the object-oriented graphics layer can insulate the rest of the system from most of the differences between these two window managers. Of course, the users of Postscript will have additional functionality, which is why a Postscript interface is planned.

The graphical objects that will be supported initially include rectangles, lines, polygons, polylines (multiple, connected lines), circles, circular arcs, lines with arrowheads, and "roundtangles" (rounded corner rectangles). Postscript provides splines, so these will be added later. Any of the closed graphical objects can be filled with a color or halftone shade, and the edges of any object can be drawn with various colors, thicknesses and patterns. Each graphical object contains a number of "slots" or values. These determine the specific properties of that object. For example, the slots for a rectangle include `left`, `top`, `width`, `height`, `filling-color`, and `outline-thickness`. These slots can be filled with a specific value or a formula (see section 5.2).

Graphical objects can be created, displayed, erased, moved, and modified. When they are displayed, they remember their properties so if they are modified in any way, they can erase the old version and redisplay themselves correctly in the new place. The designer does not have to worry about any of this mechanism; any property of a graphical object can simply be changed, and the correct result will appear on the display. This same mechanism also handles window refresh automatically (when a window becomes uncovered).

When an object is erased, many other objects may need to be redisplayed. This is because the object may be overlapped by the other objects and objects can be drawn using a variety of drawing modes. For example, if object A is drawn on top of object B using DRAW mode (OR), when object A is erased, it will leave a hole in object B (see Figure 2). Therefore, B must be redrawn after A is erased. The object-oriented graphics system handles this automatically. As a special optimization, if an object is drawn using XOR mode, and there are no other objects on top of it that were not drawn with XOR, then the object

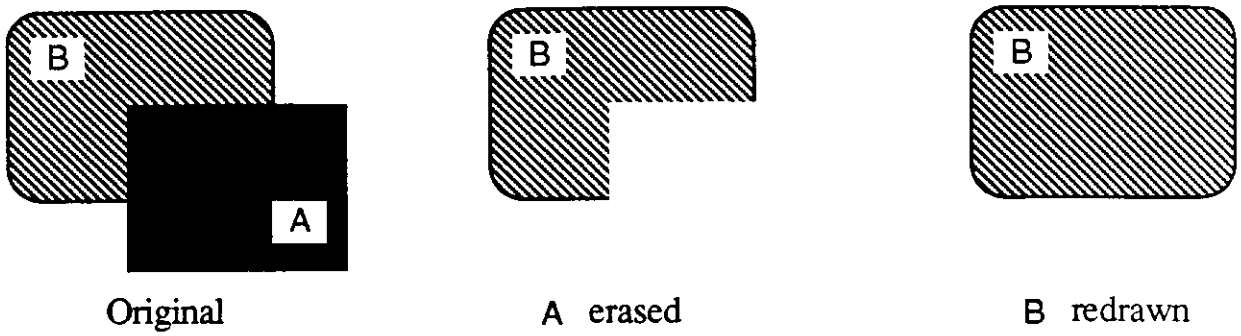


Figure 2.

When object A is erased, object B must be redrawn or else a hole will be left in it.

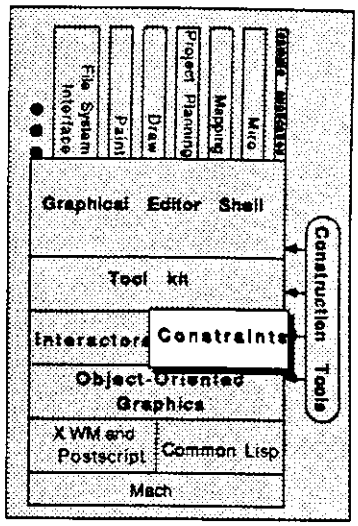
is simply redrawn using XOR mode to erase it. This will remove the object without affecting any other objects, and is therefore much faster. XOR mode is often used for objects that follow the mouse and therefore need rapid redisplay.

One important aspect of the graphical object layer is that there are special graphical objects called *aggregates* which are used to hold a collection of other objects (either primitives or other aggregates). Aggregates are used to control refresh and picking (deciding which object should be selected when a mouse button is hit).

When objects are erased, aggregates can limit the number of other objects that are redrawn. For example, the designer can use a special class of aggregate when the sub-objects do not overlap. This can be used for menus and some other structured objects. For these, only the object that is changed or erased ever needs to be redrawn. This can also be used for limiting the various screen regions that are refreshed. For example, if a window is divided into various menu and control areas and a main drawing area, then changes in the drawing area will not cause the control areas to be redrawn. We also expect there to be "smart" aggregates that will use complex window-manager-style algorithms and clever clipping on erase operations to minimize the number of objects that need to be redrawn.

It is important to note that although this document discusses only graphical objects, the object system and the constraint system are entirely general-purpose. Therefore, it should be straightforward to have objects that control external lights or special displays, or even *sound* objects that cause various noises, music or speech. These could be tied to the rest of the user interface and the application program using the existing constraint system without change.

5.2 Constraints



One of the primary foundations of the Garnet UIDE is a constraint system. This part is more fully described in a conference paper [Szekely 88]. As explained above, a constraint is a relationship among objects that is automatically maintained by the system. Garnet uses constraints to connect various graphical objects together (e.g., a string is centered inside a rectangle), to connect graphical objects with the input devices (e.g., the rectangular indicator in a scroll bar is constrained to follow the mouse while the left button is pressed), and to connect the user interface to application programs through data constraints (e.g., the displayed value on the screen is constrained to be the same as the temperature value in the application).

Constraints are used in Garnet because they are a natural way to express the kinds of relationships that are common in user interfaces. For example, an application program might want an arrow to stay connected to a rectangle even if the rectangle is moved. With constraints, the system insures that all dependencies are updated appropriately. In Peridot [Myers 88], constraints proved to be a very effective way to specify user interface relationships.

The constraint system in Garnet is called Coral, which stands for Constraint-based, Object-oriented Relations And Language, and is composed of two parts: a declarative language that makes specifying the constraints easier, and a constraint satisfaction mechanism that actually enforces the constraints.

The declarative language allows a designer to easily declare that a group of objects are related. Any slots of an object can have as their value a *formula* that references other objects. A formula can be an arbitrary Lisp expression, and the references to other objects use the function `sv` (for "slot-value"). The form for these references is `(sv <object-reference> <slot-name>)`. The `<object-reference>` can either be a specific object, or it can be a variable into which an object is later assigned.

Allowing formulas to refer to objects using variables is an important feature of Garnet. This is used for defining constraints in the abstract and later applying them to different objects. For example, a feedback rectangle might be constrained to be the size of whatever it is over. Only one feedback rectangle with one set of constraints is needed, and when the object referred to is changed, the size of the rectangle changes automatically.

As an example of the declarative language, the code in Figure 3 creates a labeled button with a check mark. This example is taken from [Szekely 88], but the specific syntax has changed somewhat.

Another aspect of the declarative language is that constraints can be defined over sets of objects. This is useful for menus and other lists. For example, there is a simple syntax for specifying that each menu


```
(defgob Check-Box (viewport label-string left-x top-y)
  (:is-a :Aggregate)
  (:viewport viewport) ; Window to display in
  (:selected NIL) ; This controls whether this box shows the check.
  (:left left-x)  ; Position of the check-box,
  (:top top-y)    ; Size will be sum of all components.
  (:parts
        (Box      ; Definition of the frame.
            (:is-a :Rectangle)
            (:outline-thickness 1)
            (:filling-color white)
            (:left (sv Check-Box left))
            (:top (sv Check-Box top))
            (:width 15)
            (:height 15))
        (Label    ; Definition of the label
            (:is-a :String-GOB)
            (:left (+ (sv Box :right) 8))
            (:center-y (+ (sv Box :center-y) 2))
            (:the-string label-string))
        (Mark     ; Definition of the check mark and its parts
            (:is-a :Aggregate)
            (:visible (sv Check-Box selected)) ;whether mark is shown or not
            (:parts
                (Line-1
                    (:is-a :Line-GOB)
                    (:x1 (- (sv Box :center-x) 2))
                    (:y1 (- (sv Box :bottom) 3))
                    (:x2 (- (sv Box :right) 2))
                    (:y2 (+ (sv Box :top) 2)))
                (Line-2
                    (:is-a :Line-GOB)
                    (:x1 (+ (sv Box :left) 2))
                    (:y1 (+ (sv Box :center-y) 1))
                    (:x2 (sv Line-1 :x1))
                    (:y2 (sv Line-1 :y1)))))))
```

Figure 3.

Definition of a single `Check-Box` demonstrating the use of the declarative language for specifying constraints. At the top are two instances of `check-box`.

item except the first should be centered underneath the previous, or that the width of a box should be the maximum of the widths of all the objects inside it.

The implementation of constraint satisfaction is designed to be very efficient. Constraints in Garnet are "one-directional." This means that there always must be at most one formula for any slot. The result of this is that there can never be a choice about how to solve a constraint, so no planning is necessary. Therefore, when an object changes, the system always knows immediately which other objects to change, and how to change them. Surprisingly, this restriction does not substantially limit the interfaces that can be created, because it is almost always possible to find a one-directional way to specify any group of constraints.

This restriction does not prevent circular constraints. For example, object A can depend on object B, and object B can depend on object A. This works by having an initial value for every slot, and if a constraint would cause a slot to be evaluated that already was given a value, then this slot is not reassigned. Therefore, if object A is changed, object B will be updated and visa-versa. It is allowable for the formulas to assign different values than are already in a slot, but then they are ignored. For example, if A is `(B + 1)`, and B is `(A + 1)`, then if B is set to 4, A will get 5, and if A is set to 4, then B will get 5. The reverse constraint is simply ignored in each case. This kind of constraint is not likely to be used in practice, since normally the constraints will just insure that A and B have consistent values.

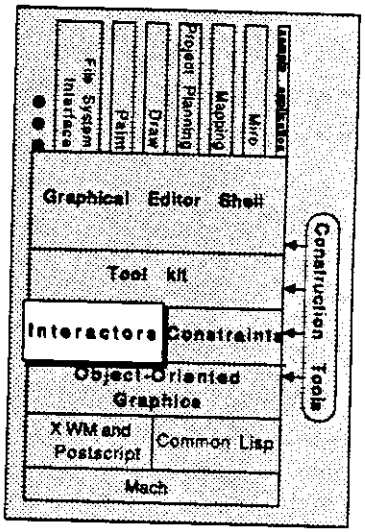
Although the constraints are designed to be used for graphical objects, they are implemented in a general-purpose manner, and can therefore be used by applications for defining relationships on their own data, if desired. Since they are efficiently implemented, applications may find it convenient to use constraint for maintaining data integrity and dependencies, for example.

In summary, the important aspects of the use of constraints in Garnet are:

- there is an easy-to-use language for specifying constraints,
- constraints can be defined in the abstract and dynamically attached to different objects,
- constraints can be defined on lists of objects, and
- constraints are implemented in an efficient manner.

5.3 Interactors

One of the most difficult tasks when creating highly-interactive user interfaces is handling the mouse, keyboard and other input devices. Typically, window managers or user interface toolkits only provide a stream of mouse positions and key events and require that the programmers handle all interactions themselves. Garnet tries to provide significantly more help through the use of *interactors* which are encapsulations of input device behaviors. The observation that makes this feasible is that there are only a small number of different kinds of behavior that are used in user interfaces. For example, although the graphics can vary significantly and the specific mouse buttons used may change, most menus operate in the same manner. Another example is the way that objects move around when following the mouse. Garnet captures these common behaviors in a central place while still providing a high degree of customizability to application programs.



Another advantage of the use of interactors is that it helps to separate and modularize the user interface software. The graphics are defined using the object-oriented graphics package and constraints, and the interactive behaviors are programmed separately using interactors. The interactors are connected to the graphics using constraints.

Interactors also provide a level of window manager independence. The designer is freed from details of how events are queued and how exception conditions are presented. The object-oriented graphics package and the interactors should provide a complete layer hiding the details of the window manager. This should allow applications to be easily ported to various window managers.

In addition, interactors directly support multiple input devices active at the same time. For example, one interactor might be handling input from the mouse at the same time another is handling a knob or touch tablet. The application program can be completely unaware of this parallelism because it is handled internally by the interactors. Research has shown that people can effectively and easily use two hands to provide such parallel input and thereby execute certain tasks much quicker [Buxton 86]. Special interactors could be built for these other input devices. For example, there might be one for one-dimensional changes to handle sliders and knobs.

All of the interactors are parameterized in various ways. For example, most interactors allow the designer to specify which mouse button or keyboard key causes them to start operating, and which causes them to stop. The standard window manager keyboard mappings will probably be used so that applications will define their actions on abstract events, and the end user can easily change which specific keyboard keys or mouse buttons are mapped to these events.

More importantly, parameters to an interactor include the particular graphics that are used for feedback and as the active areas. This means that interactors are independent of specific graphical representations on the screen. For example, a parameter to the menu interactor is the feedback graphics to show which item is selected. The user interface designer can have this be an outline box, a reverse-video rectangle, an arrow, or whatever his or her imagination can invent, and the same interactor is used without modification (see Figure 4). The interface between the graphics and the interactors is through abstract constraints, so the graphics and interactors can be easily defined and edited separately.

In designing the interactors, there are many tradeoffs that have to be considered. The current design attempts to balance flexibility and power with ease-of-use. The earlier Peridot system [Myers 87] only had very simple interactors, and multiple interactors were needed for many common operations. For example, to have an outline box follow the mouse while a button is pressed and have an object jump to the final position when the mouse button is released required three interactors: one to control the visibility of the feedback object, one to have it track the mouse, and one to have the actual object jump to the final position. In Garnet, interactors are higher level so that this behavior is achieved with one interactor. A result of this is that individual interactors have more parameters (to control the various options for feedback), and there are a few more interactor types than in Peridot.

An important design goal, however, was to limit the number of different types of interactors provided by Garnet. There are only six types and these handle almost all interactive behaviors in user interfaces. Since these are implemented in an object-oriented fashion, it is relatively easy to create new interactor types, but this is not expected to be necessary (except to support entirely new kinds of interactive

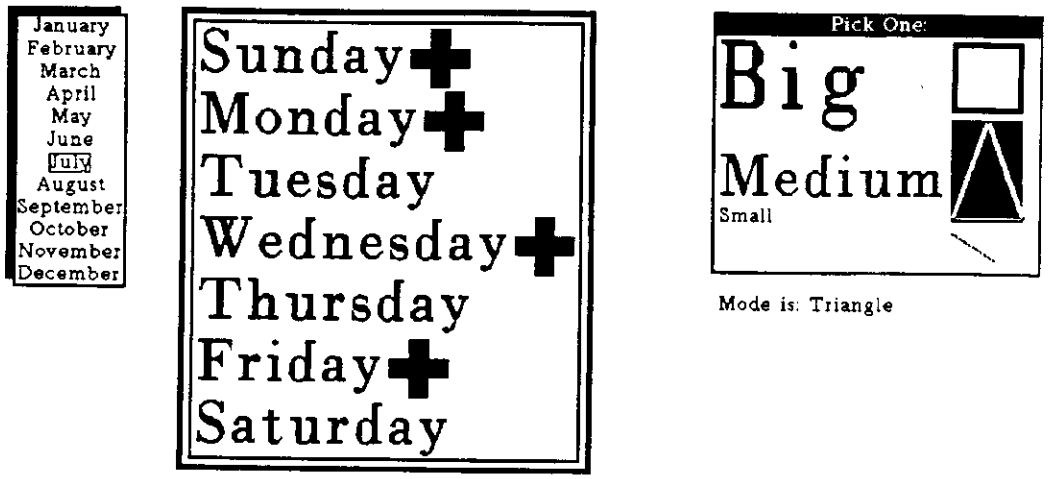


Figure 4.

The same interactor type can be used for menus with very different feedback and background graphics. Here, an outline box, plus signs, and a reverse-video rectangle (over the triangle) are being used as the feedback.

behavior in the future, such as gesture and character recognition, or new input devices). The types of interactors that are planned for Garnet include:

`Menu-Interactor` - A feedback object (e.g., a reverse video rectangle) is shown over the item underneath the mouse, and the feedback follows the mouse. This is useful when the mouse can be used to chose among a set of items, as in a menu.

`Button-Interactor` - A feedback object (e.g., a check mark) is shown for the object under the mouse when the button is first pressed, and the feedback goes on and off as the mouse moves over and away from that one object. The difference from `Menu-Interactors` is that with menus, when the mouse moves over a different object, the feedback moves to that object, but with `Button-Interactors` the feedback does not move to other objects. When the mouse moves away from the initial object, the feedback is turned off, and when it moves back, the feedback is turned back on. This interactor is useful for sets of buttons like "radio buttons" and "check boxes," and also for single, stand-alone buttons.

`Select-and-Move-Interactor` - This is used to move an object or one of a set of objects using the mouse. There may be feedback to show where the object will be moved, or the object itself may move with the mouse. When there is feedback, the object jumps to the mouse position when the interaction is complete (e.g., the mouse button is released). This interactor can be used for one-dimensional or two-dimensional scroll bars, and also for moving application objects in a graphics editor. This interactor can also be used just to select the object under the mouse by having the interactor return immediately when the start event happens.

`Select-and-grow-Interactor` - This is similar to `Select-and-Move-Interactor` except that it changes an object's size instead of its position. It is useful for graphics editors and horizontal and vertical gauges.

`Two-Point-Interactor` - This is used when either one or two new points is desired from the mouse. A rubber-band feedback object (usually a rubber-band line or rectangle) may be drawn based on the points specified.

`Angle-Interactor` - This is used to get the angle the mouse moves around some point. It can be used for circular gauges or for "stirring motions" for rotating.

`Text-string-Interactor` - This is used to input a one-line edited string of text.

To get an action to happen when a single keyboard key is hit or when a mouse button is hit over a particular object, the `Select-and-Move-Interactor` can be used, since the start action for an interactor can be any keyboard key or button.

As a specific example of an interactor, Figure 5 gives the code for a `Button-Interactor` that might be used with the check box of Figure 3.

```
(create-interactor NIL :button-interactor
        (:window mywindow)
        (:continuous T)
        (:start-where 'Check-Box)
        (:start-event :left-down)
        (:stop-event :left-up)
        (:how-set :toggle))  ; each time it toggles whether the check mark
                             ; is shown or not.
        ;; By default, the interactor sets the :selected field of the
        ;; :start-where object, which is the desired behavior.
```

Figure 5.

Code for an interactor to control the check box of Figure 3.

In general, there will be an instance of one of these interactor types for each menu, each set of buttons, and each set of objects to be moved. For example, for a scroll bar with two arrow icons like on the Macintosh, there might be three interactors: one `Button-Interactor` for each icon, and one `Select-and-Move-Interactor` to allow the indicator to be moved. Allowing interactors to work on a set of objects (rather than just on one individual object) will help significantly decrease the number of interactors that are needed. For example, one interactor can handle a menu (rather than one interactor per menu item), and one interactor can handle an entire set of objects that can be individually moved with the mouse in a graphics editor.

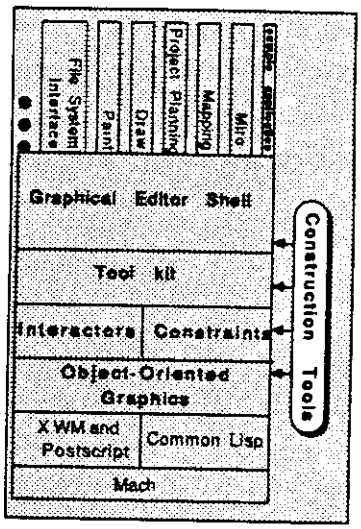
Each interactor has a default action when starting, running, and stopping, but the designer can also provide arbitrary lisp procedures to be called at appropriate points. This seems to provide an appropriate balance between ease of use (using the defaults) and flexibility (writing procedures). Even when procedures are needed, using interactors is still useful because they handle the control of sequencing and provide a layer of device and window manager independence.

In summary, the important features of the interactors in Garnet are:

- They provide a level of abstraction when defining user interfaces since the behavior of the input devices can be specified at a high level and independently from the graphics.
- They identify six basic behaviors common to most highly-interactive user interfaces and implement these in a single module. It appears that almost all typical user interfaces can be created out of these six types of interactors.

5.4 Construction Tools

Most highly-interactive user interfaces have significant portions that are pictorial. The Garnet construction tools will allow the user interface designer to specify all of the pictorial parts of the user interface by actually drawing them graphically. The appropriate code to create these pictorial parts will then be generated automatically.



The primary tool will be a form of graphical editor that will allow pre-defined interaction techniques, such as menus, scroll bars and buttons, to be placed in windows. This tool is called Lapidary (a Lapidary is a person that cuts and polishes precious stones, including Garnets).[2] In addition to placement, however, Lapidary will allow the graphics of these interaction techniques to be edited. Therefore, if a new kind of menu or a different look and feel for buttons is desired, they can simply be drawn using this tool. The Peridot system demonstrated that this is possible, but Garnet extends Peridot by making it easy to use pre-defined interaction techniques. Therefore, Garnet could be used to create a tool kit of interaction techniques that conform to a particular "corporate style," and then Garnet could also be used to create user interfaces using the tool kit.

Other tools that might be provided include an icon editor, a font editor, a keyboard mapping editor, a menu sequence editor (for deep menu hierarchies), etc.

Another important use for the construction tools will be to build objects that the application will manipulate at run-time. For example, if the designer wants to cause a box with a particular label to appear when the end user presses a button, the designer can create the box using the construction tools and program the control using the interactors. Most previous construction tools have been limited to the parts of the user interface that are not controlled by the application. Garnet's tools, on the other hand, attempt to cover every pictorial part of the user interface.

---

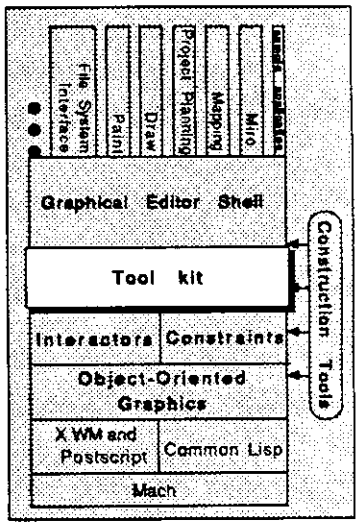
[2]Lapidary was chosen by the author's colleagues to frustrate his acronym-making efforts and currently does not stand for anything.

The other parts of Garnet were designed to make it easy to create the construction tools, based on experience with Peridot. For example, it is often the case that an interaction technique should change size in particular ways when the window it is in changes size; e.g., most vertical scroll bars change height vertically but stay the same width horizontally. Other systems have used special tricks to allow this to be specified [Cardelli 87]. In Garnet, however, the general constraint mechanism handles this easily. Another example is that the interactors are parameterized by the kinds of things that the user interface designer is likely to want to change about the user interface: the particular buttons that cause interactions to start and stop, and the specifics of the graphics to be used for the various parts of the interaction.

Unlike Peridot, the Garnet construction tools will probably not use plausible inferencing to guess the constraints desired in the user interface. Although this "Programming by Example" was fairly successful in Peridot, it was decided to attempt a more "direct manipulation" approach in Garnet and require all relationships to be explicitly specified. For example, whereas Peridot attempted to guess the graphical constraints among the objects drawn, the Garnet tools will allow the user to explicitly define the constraints using iconic menus. We expect the construction tools themselves to be very easy to use, and may even be usable by non-programmers.

5.5 Tool kit

The tool kit component of Garnet is primarily a demonstration that the other parts work, and will be a starting point for designers using the system. The tool kit will contain a collection of menus, scroll bars, buttons, and other interaction techniques that are attractive and well-integrated. The user interface designer can either use these interaction techniques directly or edit them to get different looks and behaviors using the construction tools.

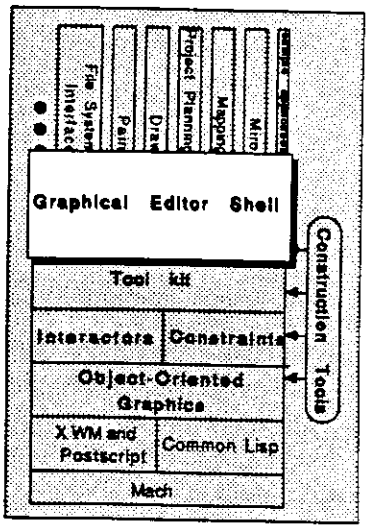


The tool kit itself will be built using the constraints and interactors to demonstrate that they are usable. It is expected that significant parts of the tool kit will be constructed with the graphical construction tools, and that all of the tool kit items will be accessible and editable using those tools.

5.6 Graphical Editor Shell

Many graphical, highly-interactive programs have a large number of similarities. Garnet's "Graphical Editor Shell" is designed to handle many of these features in a central place. For example, it might handle:

- Selecting application objects,
- Creating new objects when the end user presses on a palette,
- Moving objects and changing their size using the mouse,
- Giving commands from menus and from the keyboard,
- Updating the screen when there are changes in application data structures which are associated with visible objects, or when the end user changes an object,
- Undoing of commands,
- Providing help for commands and functions, and
- Printing of the picture at high resolution on laser printers.



The Graphical Editor Shell is envisioned as a highly-customizable graphical editor that the designer will be able to change in various ways. There will be various options for all of the above features, so the designer can get the desired behavior and appearance. In addition, the designer can define constraints to make the appropriate application-specific restrictions.

While it is clear that the Graphical Editor Shell will not handle every kind of user interface that might be desirable, we expect it to have a fairly broad range. Some examples of the kinds of programs it might be able to create include:

- Conventional drawing programs such as Apple Macintosh MacDraw and MacPaint,
- The Macintosh Finder, which allows file manipulations by moving icons around (rename, delete, change directory, etc.),
- Project planning programs such as Apple's MacProject, where activities are boxes and dependencies are shown as arrows,
- Graphical Programming Languages [Myers 86b, Tygar 87] where computer programs can be constructed using icons and other pictures (a common example is a flowchart),
- User interface construction tools, including all of the ones listed in section 5.4,
- Chess and other board game user interfaces,
- Simulation programs, where the user interface visualizes the status of the simulation, and
- Tree and graph editing programs, including semantic networks, neural networks, state transition diagrams, etc.

While it is clear that we will not be implementing the application-specific algorithms that make these user interfaces different (for example, we are not planning to implement any simulation systems or automatic graph layout packages), we expect that the Graphical Editor Shell will be general enough to make programming the user interfaces for all these systems significantly easier.

6. Implementation Issues

An important implementation consideration for Garnet is that it should be "fast." For a user interface, this means that objects being dragged with the mouse should appear to stay attached and not lag behind, and that animations should happen without noticeable jumping and flicker. We expect that Garnet will be able to meet these requirements.

Since Garnet is a research project, another important consideration is that we chose the implementation environment that will allow us to make progress in the most expeditious manner possible. Therefore, we decided to use Lisp as the programming language. This makes it easy for Garnet to generate code and then execute it immediately, which is necessary for the constraints and for the construction tools. It is also generally easier to generate experimental code in Lisp rather than in compiled languages [Sheil 83].

On the other hand, it is desirable to have other people use our software and to implement it in a portable way. Therefore, we have chosen to use CommonLisp [Steele 84] since it has been implemented on a variety of machines. Because there is a high degree of local support and general availability around Carnegie Mellon, we are using CMU CommonLisp [McDonald 87].

The same portability arguments suggest that we should use a standard window environment, and so Garnet is being built on top of the X window manager [Scheifler 86]. X has significant performance problems, however, and it does not support many graphics functions that we need. For example, X version 11 does not support splines. Therefore, we hope to be able to use the NeWS window manager from Sun [Sun 86] or possibly Display Postscript from Adobe. Both of these will be available integrated with X, so we will be evaluating whether that will provide an appropriate platform for further Garnet work. In the future, it might be interesting to see if Garnet can be ported to other CommonLisp implementations, such as on the Apple Macintosh or Microsoft's OS/2. In any case, Garnet's interface to applications should not be affected much since the window manager's interface is hidden from programs.

Another issue was which object system to use on top of CommonLisp. The obvious choice is the Common Lisp Object System (CLOS) which is an emerging standard. After about six months of evaluation of CLOS, however, we decided that it did not meet our needs. A significant problem is speed. The measured time to access a class variable in CLOS was up to 10 times slower than to access the field of a structure (CommonLisp records). Another important problem is that a preliminary design for the construction tools uses a form of objects that are incompatible with CLOS. With CLOS, there are classes and instances of those classes. To get a new type of object, you have to make a new class and then instances of that class. The construction tools, however, need to be able to make many different types quickly and to be able to edit the structure of instances. This suggests that a better object system for Garnet would be one where any instance can be used as the "prototype" for other instances. Therefore, when a new instance is made, an existing instance is used as the prototype and all variables and methods of the parent instance are inherited by the child instance unless overridden. This makes the object hierarchy much more dynamic and easily edited. The KR system [Giuse 87] provides this flexibility and is being used as the Garnet object system.

7. Validation

Although Garnet is being built as a research project, we still plan to make it usable. In order to verify that the ideas are workable, we will try to have a small number of designers using it. First, the higher layers of Garnet will be built using the lower layers. For example, the tool kit will be built using the interactors, constraints, and graphical object system. The construction tools will also be built using these components. The Graphical Editor Shell will be built using all of the lower layers. Significant parts of the construction tools will be created using the Graphical Editor Shell. We will also build a few demonstration applications on top of the Graphical Editor Shell, possibly including a simple graphical editor and a simple "Finder." Of course, Garnet will create its own user interface.

In addition to the internal users, at least one large outside project is expected to use Garnet. The Miro group [Tygar 87] is developing a graphical programming language for defining security relationships in operating systems. Garnet will be used by them to create a graphical editor for their visual language. Once Garnet is more mature, we expect that there will be additional outside users.

8. Conclusions

In conclusion, Garnet is a very exciting research project investigating the software design for user interfaces. The various parts of Garnet have important unique aspects, and the integration of the parts will create a system that is significantly different from any now existing. In summary, the contributions of Garnet include:

- Demonstrating that constraints can be efficiently and effectively used to build user interfaces.
- Identifying a small number of interactors which encapsulate interactive behaviors, and demonstrating that significant user interfaces can be created easily using them.
- Demonstrating that all graphical aspects of user interfaces can be specified using graphical construction tools instead of by programming.
- Identifying common functionality for a wide class of graphical, highly-interactive user interfaces, and designing a way to provide this in a central "Graphical Editor Shell," while still providing the needed flexibility for applications.
- Demonstrating that significant user interfaces can be created using these technologies, and therefore that they are practical for use in real systems.

We expect that the Garnet approach to user interface software design will be successful, and may therefore significantly impact research in this area and possibly change the way that future commercial user interface software is built.

Acknowledgements

I would like to thank Duane Adams, Nico Habermann, Scott Fahlman and the CMU CS department for supporting the Garnet project. For help with this paper, I want to thank Roger Dannenberg, Dario Giuse, Brad Vander Zanden, and Bernita Myers.

References

[Barth 86] Paul Barth.
An Object-Oriented Approach to Graphical Interfaces.
ACM Transactions on Graphics 5(2):142-172, April, 1986.

[Bobrow 86] Daniel G. Bobrow, Sanjay Mittal, and Mark J. Stefik.
Expert Systems: Perils and Promise.
Communications of the ACM 29(9):880-894, September, 1986.

[Borning 79] Alan Borning.
Thinglab--A Constraint-Oriented Simulation Laboratory.
Technical Report SSL-79-3, Xerox Palo Alto Research Center, July, 1979.

[Buxton 83] W. Buxton, M.R. Lamb, D. Sherman, and K.C. Smith.
Towards a Comprehensive User Interface Management System.
In *Computer Graphics*, pages 35-42. SIGGRAPH'83, Detroit, Mich, July, 1983.

[Buxton 86] William Buxton and Brad Myers.
A Study in Two-Handed Input.
In *Human Factors in Computing Systems*, pages 321-326. SIGCHI'86, Boston, MA, April, 1986.

[Cardelli 87] Luca Cardelli.
Building User Interfaces by Direct Manipulation.
Technical Report 22, Digital Equipment Corporation Systems Research Center Research Report, October, 1987.

[Giuse 87] Dario Giuse.
KR: an Efficient Knowledge Representation System.
Technical Report CMU-RI-TR-87-23, Carnegie Mellon University Robotics Institute, October, 1987.

[Goldberg 83] Adele Goldberg and Dave Robson.
Smalltalk-80: The Language and Its Implementation.
Addison-Wesley Publishing Company, Reading, MA, 1983.

[McDonald 87] David B. McDonald, editor.
CMU Common Lisp User's Manual.
Technical Report CMU-CS-87-156, Carnegie Mellon University Computer Science Department, September, 1987.

[Meads 87] Jon Meads.
The Standards Factor.
SIGCHI Bulletin 19(1):34-35, July, 1987.

[Mittal 86] Sanjay Mittal, Clive L. Dym, and Mahesh Morjaria.
Pride: An Expert System for the Design of Paper Handling Systems.
IEEE Computer 19(7):102-114, July, 1986.

[Myers 86a] Brad A. Myers and William Buxton.
Creating Highly Interactive and Graphical User Interfaces by Demonstration.
In *Computer Graphics*, pages 249-258. SIGGRAPH'86, Dallas, Texas, August, 1986.

[Myers 86b] Brad A. Myers.
Visual Programming, Programming by Example, and Program Visualization; A Taxonomy.
In *Human Factors in Computing Systems*, pages 59-66. SIGCHI'86, Boston, MA, April, 1986.

[Myers 87] Brad A. Myers.
Creating Interaction Techniques by Demonstration.
IEEE Computer Graphics and Applications 7(9):51-60, September, 1987.

[Myers 88] Brad A. Myers.
Creating User Interfaces by Demonstration.
Academic Press, Boston, 1988.

[Myers 89] Brad A. Myers.
Tools for Creating User Interfaces: An Introduction and Survey.
to appear in: *IEEE Software* , January, 1989.
Also available as: Carnegie Mellon University Computer Science Department Tech Report number: CMU-CS-88-107, Jan 1988.

[Olsen 87] Dan R. Olsen, Jr., ed.
ACM SIGGRAPH Workshop on Software Tools for User Interface Management.
Computer Graphics 21(2):71-147, April, 1987.

[Palay 88] Andrew J. Palay, *et. al.*
The Andrew Toolkit - An Overview.
In *Proceedings Winter Usenix Technical Conference*, pages 9-21. Winter Usenix, Dallas, Tex, February, 1988.

[Pfaff 85] Gunther R. Pfaff (editor).
User Interface Management Systems.
Springer-Verlag, Berlin, 1985.

[PHIGS 84] *Draft Proposal American National Standard for the Functional Specification of the Programmer's Hierarchical Interactive Graphics Standard (PHIGS)*
American National Standards Committee X3H3/84-40, 1984.

[Rashid 88] R. Rashid, A. Tevanian, et. al.
Machine Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures.
IEEE Transactions on Computers 37(8):896-908, August, 1988.

[Scheifler 86] Robert W. Scheifler and Jim Gettys.
The X Window System.
ACM Transactions on Graphics 5(2):79-109, April, 1986.

[Schmucker 86] Kurt J. Schmucker.
MacApp: An Application Framework.
Byte :189-193, August, 1986.

[Schulert 85] Andrew J. Schulert, George T. Rogers, and James A. Hamilton.
ADM-A Dialogue Manager.
In *Human Factors in Computing Systems*, pages 177-183. SIGCHI'85, San Francisco, CA, April, 1985.

[Sheil 83] Beau Sheil.
Power Tools for Programmers.
Datamation 29(2):131-144, February, 1983.

[Shneiderman 83] Ben Shneiderman.
Direct Manipulation: A Step Beyond Programming Languages.
IEEE Computer 16(8):57-69, August, 1983.

[Smith 82] David Canfield Smith, Erik Harslem, Charles Irby, and Ralph Kimball.
The Star User Interface: an Overview.
In *Proceedings of the 1982 National Computer Conference*, pages 515-528. AFIPS, 1982.

[Steele 84] Guy L. Steele, Jr. (editor).
Common Lisp: The Language.
Digital Press, 1984.

[Sun 86] *NeWS Preliminary Technical Overview*
Sun Microsystems, Inc., 1986.
2550 Garcia Avenue, Mountain View, CA 94043.

[Sutherland 63] Ivan E. Sutherland.
SketchPad: A Man-Machine Graphical Communication System.
In *AFIPS Spring Joint Computer Conference*, pages 329-346. 1963.

[Sutton 78] Jimmy A. Sutton and Ralph H. Sprague, Jr.
A Study of Display Generation and Management in Interactive Business Applications.
Technical Report RJ2392, IBM Research Report, November, 1978.

[Szekely 88] Pedro A. Szekely and Brad A. Myers.
A User Interface Toolkit Based on Graphical Objects and Constraints.
In *ACM Conference on Object-Oriented Programming: Systems Languages and Applications.* OOPSLA'88, San Diego, CA, September, 1988.

[Tygar 87] J. D. Tygar and J. M. Wing.
Visual Specification of Security Constraints.
In *1987 Workshop on Visual Languages*, pages 288-301. Visual Language'87, Linkoping, Sweden, August, 1987.

[Vander Zanden 88] Brad T. Vander Zanden.
Constraint Grammars in User Interface Management Systems.
In *Proceedings Graphics Interface*, pages 176-184. GI'88, Edmonton, Canada, June, 1988.

[Williams 83] Gregg Williams.
The Lisa Computer System.
Byte Magazine 8(2):33-50, February, 1983.