Building User Interfaces by Direct Manipulation

Luca Cardelli

Digital Equipment Corporation, Systems Research Center
130 Lytton Avenue, Palo Alto, CA 94301

Appears in: ACM Siggraph Symposium on User Interface Software, pp. 152-166, ACM Press, 1988.

SRC Research Report 22, October 2, 1987. Revised January 1, 1993.

1. Introduction

User interfaces based on mice, bitmap displays and windows are becoming commonplace, and there are guidelines on how such interfaces should function [Apple 85]. As a consequence, there is a growing expectation that all programs, no matter how trivial or how complicated, should present a graphically elegant and sophisticated user interface.

Unfortunately, such polished interfaces are normally difficult to build. There are three major factors contributing to this problem.

The first one is what we might call the *artistic burden*: the artistic insight needed in preparing good-looking interfaces. This requires taste in the choice of shapes, proportions, arrangements and colors. Professional graphic designers are sometimes assigned to work on this aspect of an interface.

The second factor is the *polishing burden*. An interface must be functional, in the sense of providing ways of carrying out tasks, but this is just a minimal requirement. Most of all, an interface must be smooth, in the sense of being intuitive and not causing surprises to the user. Smoothness is achieved by using coherent paradigms (the best known is the "desktop" paradigm [Smith 83]), by constantly redesigning and polishing interfaces as the result of user feedback [Buxton 80], and by eliminating minor interface problems which often have no effect on functionality or performance, but greatly contribute to the "professional" feel of an interface.

The third factor is the *programming burden*: the knowledge of graphics and windowing techniques required to build such interfaces. Application programmers often would rather not know all the low-level quirks of a window system. Even when a programmer is familiar with the details of the techniques, the difficulty of "getting things to work" may be such that building even a small user interface is perceived as a major task to be undertaken only for important applications. Furthermore, final polishing of the task is reserved only for those applications considered important enough to deserve the extra time. As a consequence of the programming burden, application programmers tend to invest less effort than is ideally needed, and the quality of their user interfaces degrades.

Burdens should not necessarily be eliminated. We do want interface builders to have fine control on the appearance of an interface, to be able to achieve smoothness to the desired degree, and to be able to program arbitrary behavior when needed. Our goal, however, is to make these tasks much simpler, so that application builders and even application users can confront them as routine and painless activities.

The approach described in this paper achieves this goal by separating the user interface from the application program, as is done in many *user interface management systems* [Pfaff 83], and by using a *user interface editor* to build the interfaces. In a sense, we apply the *direct manipulation* style [Schneiderman 83] characteristic of user interfaces to the very process of building them, as opposed to building them by programming. The disadvantage is that we have to determine fixed but sufficiently general classes of user interfaces that can be assembled by direct interaction and yet cover most of the common cases. However, we gain a method of easily constructing, modifying, maintaining and customizing interfaces.

1.1 Some examples

Using the tools described here, one can build user interfaces for simple applications in a very short time: for example, less than an hour for the examples below. Moreover, one can often modify such interfaces without affecting the application program. To make our discussion more concrete we use as examples, throughout the paper, three simple programs exhibiting a graphical user interface.

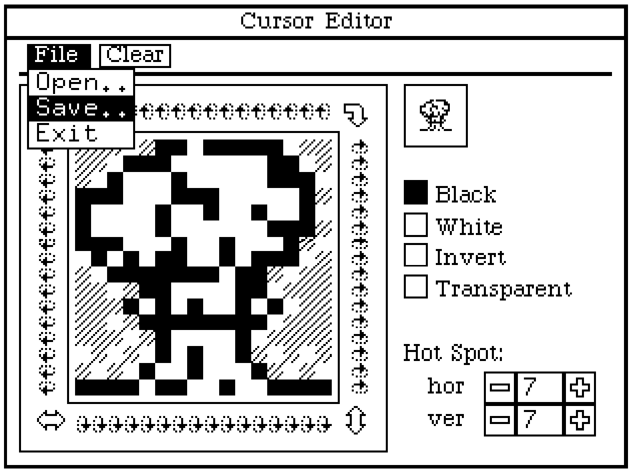
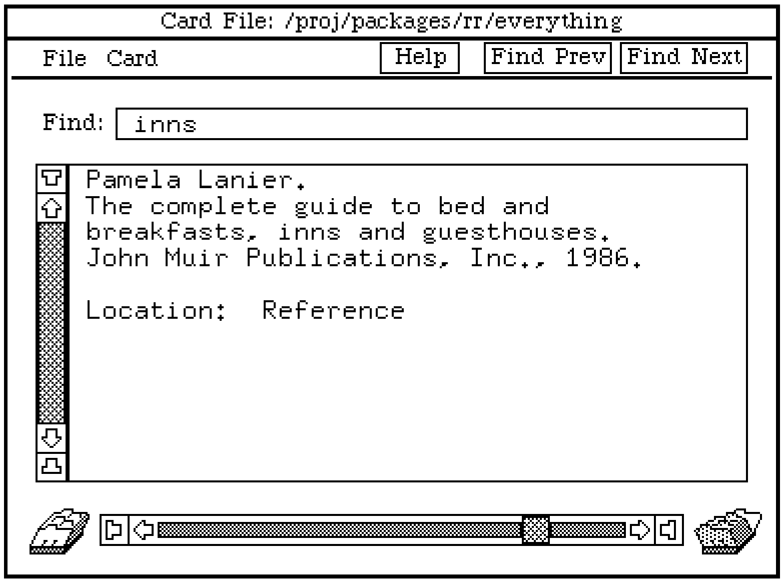


Figure 1: A cursor editor

The first program, a *cursor editor*, is used to edit the shape of the 16x16 cursor which tracks the mouse movements on a bitmap screen. The cursor editor interface in Figure 1 consists of a central, magnified, bitmap region, which can be used as a drawing pad; a real size image of the cursor; a choice of painting colors determining how every pixel modifies the background; and a "hot spot" setting specifying the pixel at which the cursor is pointing. Button icons around the central region provide rotations, reflections, and tumbling. A "File" pulldown menu is used to read and write files containing cursor images, and a "Clear" button is used to clear the cursor image.
The second program, a *card file browser*, (Figure 2) maintains files of index cards, such as may be used to store address lists or library references. The current card is shown in the central editable text region, and other cards can be selected by the scroll bar at the bottom (the card file icons next to it are just decoration). A card can be retrieved by typing a search string in the "Find:" region and pressing carriage return to search forward, or by pressing "Find Prev" and "Find Next" buttons. A "File" menu is used to load, store, and merge card files, and a "Card" menu is used to insert or delete cards.

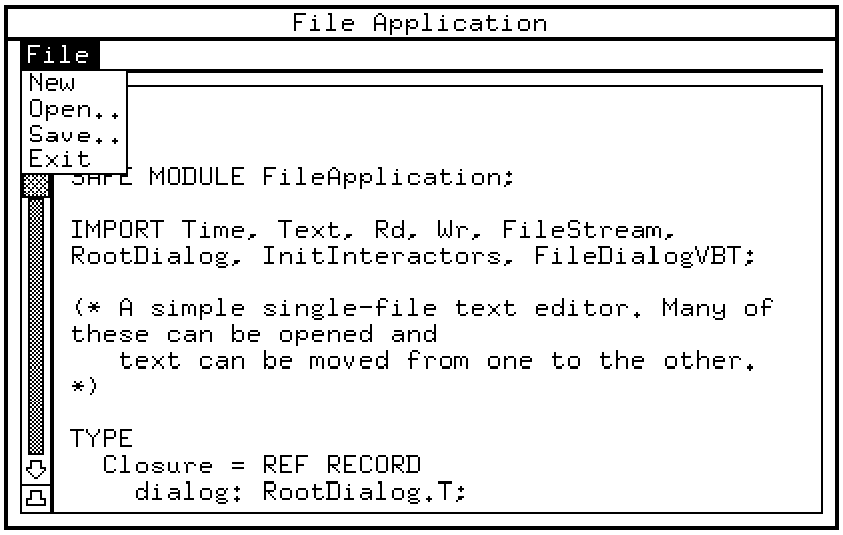


```
Pamela Lanier.
The complete guide to bed and
breakfasts, inns and guesthouses.
John Muir Publications, Inc., 1986.

Location:  Reference
```

Figure 2: A card file browser

The third program is a *single-file text editor*, shown in Figure 3. This is used in a programming example later on in the paper, and its functionality is kept to a minimum. The interface consists of a text region which can contain and edit a file, and a pulldown menu to clear the text, open and save the file, and exit.



```
SAFE MODULE FileApplication;

IMPORT Time, Text, Rd, Wr, FileStream,
RootDialog, InitInteractors, FileDialogVBT;

(* A simple single-file text editor. Many of
these can be opened and
   text can be moved from one to the other.
*)

TYPE
  Closure = REF RECORD
    dialog: RootDialog.T;
```

Figure 3: A simple text editor

These are very simple applications, and many other such applications can be imagined. As we pointed out, such applications are not necessarily simple to develop. That is, the machinery underlying a cursor editor or a card file browser can certainly be written in less than a week, but developing a customized user interface may take many days when starting from a standard user interface toolkit, or even weeks and months when starting from a bare window system.

1.2 Some principles

We now discuss some principles which we follow in order to achieve our goals. One should keep in mind that these principles may only be partially realized, but they are still useful for explaining our motivations and general orientation.

First, user interfaces should be created by direct graphical interaction through a *user interface editor*, as opposed to the common practice of generating them by calls to user interface libraries from some programming language. Direct manipulation facilitates experimentation, has quick turnaround, and results in nicer-looking interfaces. It avoids the tedious and error-prone task of describing two-dimensional layouts in linear notation; in particular it avoids specifying precise numerical positions of features or constraints between them. The main advantage of linear notation, namely the ability to parameterize to arbitrary degrees, does not seem essential in this context and, as we shall see, some useful parameterization can be achieved graphically. A user interface editor can make a program less dependent on the details of its user interface (since the interface is not described by the program) while maintaining a high degree of flexibility in customizing the interface.

Second, user interfaces should work through some *abstract protocol.* In other words, client programs should not have to know how their interfaces are put together, or understand the peculiar behaviour of interface components. For example, replacing a pulldown menu by a set of buttons should not affect the program code. Again, the intent is to make programs less dependent on the user interface. This separation of concerns facilitates quick maintenance and experimentation, since one does not have to modify (or, in our case, even recompile) an application when experimenting with the user interface.

Third, the set of user interface components should be *open-ended*. For example, we may want to provide several suites of interface components, each reflecting a different user interface style; it would then be possible to have two interfaces for the same application, exhibiting different interaction styles. Also, a new class of applications may require a new component (e.g. an analog input knob), or an individual application may require a new specialized component (e.g. a music notation editor). It should be possible to add new suites and components to the existing environment. This flexibility makes the construction of user interface editors a bit more challenging.

1.3 Interactors and dialogs

The primary function of an arbitrary application may require an interface component (e.g. for specialized graphical editing) which must be built for the occasion. However, the auxiliary functions of most applications, such as opening and closing files, can normally be carried out by a relatively small and fairly standardized set of functional components. These components, standard or ad-hoc, are here called *interactors.* Interactors include the familiar buttons, pulldown menus, scroll bars and text areas [Apple 85].

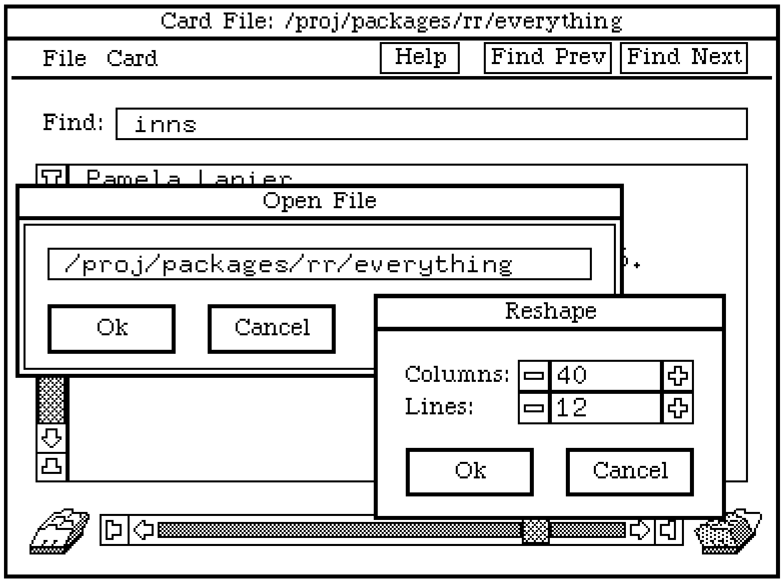


Figure 4: Pop-up dialogs

A simple user interface, or part of a complex user interface, can be realized by a small, fixed collection of interactors. We call such a collection a *dialog* (also known as a *dialog box* [Apple 85] or a *form*). Applications can use one dialog for the basic interface, and temporarily or permanently expose additional dialogs for additional functions. For example, our card file browser may pop up dialogs on top of the background dialog in response to *reshape-card* or *open-file* menu commands (Figure 4).

The *dialog paradigm* can be used as the basis for a large set of conceivable user interfaces. A "simple" user interface can be realized by a single dialog; if the interface is too complex or ill-suited for a single dialog, multiple dialogs can be used. We are not necessarily advocating using dialogs for all possible interfaces; the emphasis here is on building simple interfaces simply, without claiming that arbitrarily complex interfaces should be built this way. However, it is often the case that interfaces that are too complex to fit directly in the dialog paradigm can be redesigned to fit into it, sometimes gaining in consistency and simplicity.

The main limitation of dialogs is that they have a fixed number of interactors, each with a fixed size or, as we shall see, a size dependent on the size of their dialog. This limitation is also a feature, in the sense that it makes building dialog editors possible, since editors in general must operate on concrete and fixed data structures.

This limitation causes a problem when one needs to display large, variable-size or dynamically changing information. In such common situations several approaches are possible, if the designer wishes to remain within the dialog paradigm: (a) use interactors able to display variable-size information, for example scrollable text areas and menus; (b) animate dialogs by hiding and exposing some of their components under program control; (c) pop up additional dialogs under program control. Combinations of these ways of dealing with dynamic information turn out to be adequate in most situations.

1.4 Outline

This paper is organized around the basic pieces of information that have to be specified for any user interface; throughout, a user interface editor is used to specify such information. Section 2 shows how the *geometry* of an interface is determined by fixing the location of interface components and their geometric relationships. Section 3 shows how the *behavior* of an interface is described by the kind of interface components used and by their dynamic behavior; this behavior includes the specification of the protocol that a client program must obey to interact successfully with the interface. Section 4 shows how the *appearance* of an interface is specified by text, fonts, bitmaps, colors, etc.; similarity of appearance across interface components is captured by a notion of *graphical resources*, which can be shared by several components. Section 5 discusses how the *meaning* of an interface is determined by attaching program routines carrying out certain tasks to the *events* specified as part of the behavioral protocol. Finally, Section 6 is about the internal *architecture* of the user interface editor.

2. Geometry

In this section we discuss the geometrical aspects (shape, position and stretching) of user interface components independently of the function of such components.

2.1 Editing dialogs

Since interfaces will be organized around the dialog paradigm described in the introduction, the basic interface editing tool is a *dialog editor*. The first function of a dialog editor is to define the geometric layout of a dialog. Starting with an empty dialog, one can add instances of various kinds of interactors (which, from a geometric point of view, are all handled as rectangles), move them around and change their size, individually or in groups.

In Figure 5 we see a snapshot of the dialog editor open on our two example interfaces (the dim one in the background is inactive). The interactor near the top right of the cursor editor dialog (normally showing the cursor being edited in its actual size) has been selected and its eight *control points* have been highlighted. This interactor could now be moved around by pointing at it and dragging it, or it could be reshaped by dragging one of its control points. The central square region is a *fatbits* interactor used for editing the cursor bitmap; it is surrounded by eight buttons for rotations reflections and tumbling (which look like fat arrows or rows of little arrows). The square border around the small and fat arrows is provided by a *passive area* interactor. Passive areas can display static textures, text or bitmaps, with or without a border; the horizontal line below the "File" pulldown and the various text labels are also passive area interactors.

Each dialog has a title area at the top which can be used to drag the dialog around. In the right corner of the title area, a *reshape control* allows changing the dialog dimensions without affecting the relative positions (w.r.t. the north-west corner) of its interactors.

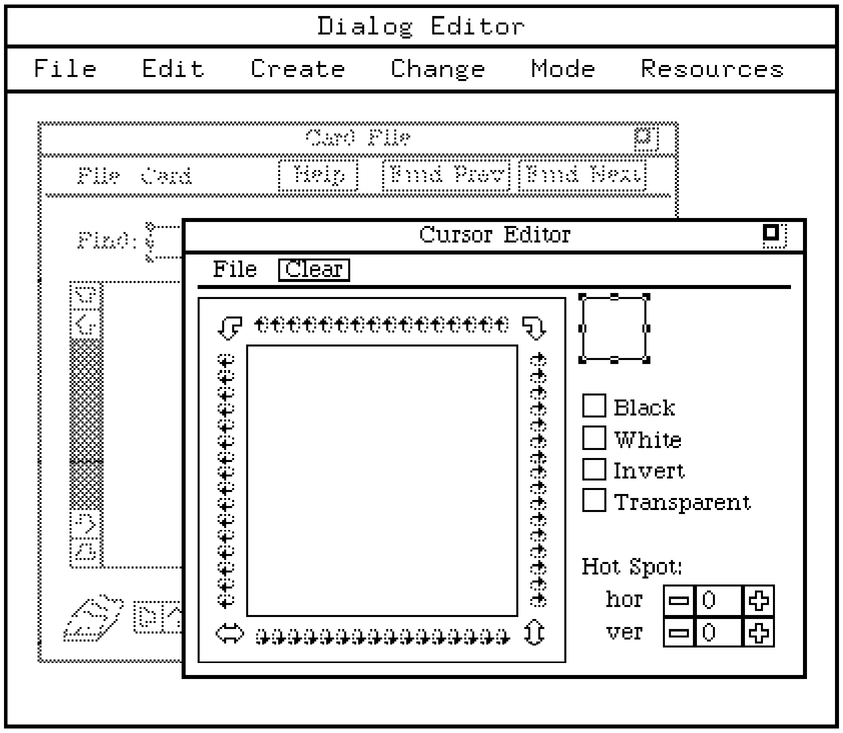


Figure 5: The dialog editor

The editing interface is similar to MacDraw [Apple 83]; one can select many items, move them around in groups, cut and paste groups, and change the shape of all the elements of a group in a single operation. A single selection is done by clicking over an item (which highlights the 8 control points). A multiple selection is done by shift-clicking over other items, or by sweeping a rectangle with the mouse: all the elements intersecting the rectangle will be selected. Moving is done by dragging items, and reshaping by dragging control points. All movements are relative to a settable grid step, for easy alignment; the grid is shown only during dragging operations.

To complete our first look at the dialog editor, here is a quick description of its pulldown menus which are shown in Figure 5 under the "Dialog Editor" header; more details will be provided in later sections. The "File" menu can be used to load and save dialog descriptions, which are stored on disk as data structures. The "Edit" menu can be used to cut and paste sets of interactors, within or between dialogs, and to move sets of interactors in front of or behind other interactors. The "Create" menu contains a list of the available interactors. The "Change" menu is used to change interactor and dialog properties. The "Mode" menu is used to switch between Edit (as in Figure 5), Stretch, Test and Debug modes; the first two have to do with geometry, the latter two with behavior. Finally, the "Resources" menu gives access to resources (e.g. fonts or colors) which are shared by many interactors in the same dialog.

2.2 Stretching

The main dialog of an application (called the *root dialog*) is embedded in a window within the window system, and hence may be subject to size changes when the user reshapes that window. Some pop-up dialogs may also be resized by the user. Hence, there should be a way of specifying how a dialog behaves under size changes.

All size changes are interpreted as *stretching* operations. A dialog has a *minimum size* which is the size of the dialog when the dialog editor is in edit mode. Dialogs are allowed to stretch, but can never become smaller than their minimum size; if they are placed in a window smaller than that, they will be clipped.

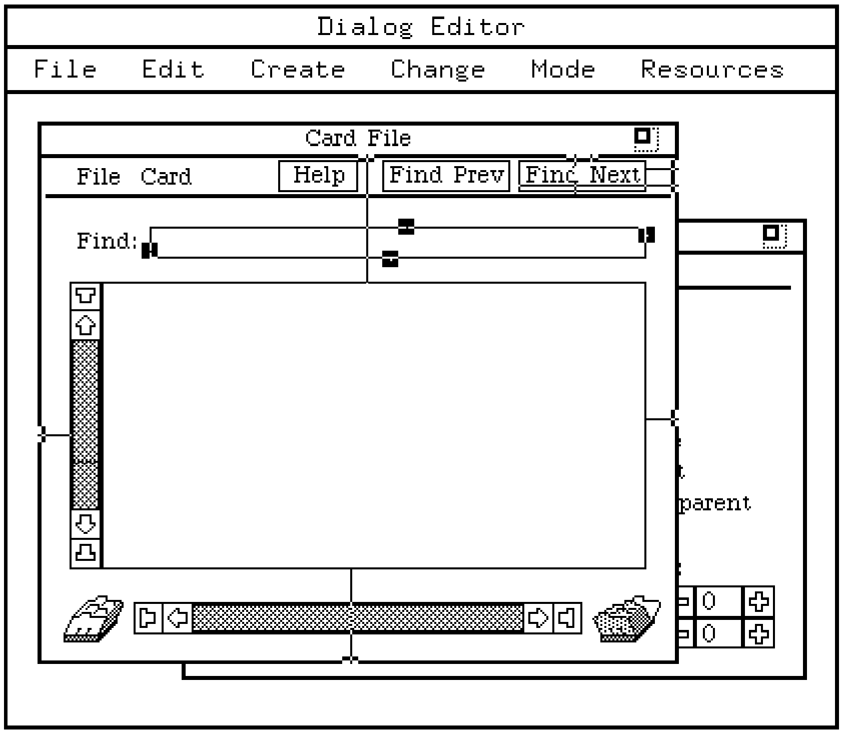


Figure 6: Specifying stretching information

Stretching is an instance of constraint resolution, and one could very easily be carried away implementing arbitrary constraints models[1]. This temptation should be resisted, for many reasons. General constraint resolution algorithms are often very complex and highly unpredictable; we want something simple and reliable, since we are not interested in overly complex geometric arrangements. More important, it may be hard to devise a graphical language for expressing general constraints. So, we have adopted a relatively simple-minded solution which allows us to express constraints graphically, but which does not always specify "correct" arrangements, in the sense of preventing abnormal interactor movements such as accidental overlapping. After all, in an interactive editing situation such problems are easily detected and fixed.

To allow stretching, each interactor has four *attachment points* which attach it to the underlying dialog, one for each edge. Attachment points are shown as black squares when the dialog editor is in stretching mode and one or more interactors are selected. Two rules

[1] Incidentally, note that unlike the TeX boxes-and-glue model, we are interested in rigid glue (preserving separation between items) and stretchable boxes (making items as large as possible).

determine the way attachment points (and hence interactors) move and stretch when the dialog is stretched.

> *First rule of stretching: as a dialog stretches from a minimum size, all the attachment points move proportionally to the stretching of the dialog.*

Figure 6 shows the card file dialog, with the editor in stretching mode; three of the interactors have been selected so that their attachment points are visible. Initially, the attachment points lie on the edge of their interactors (as in the text-line box to the right of the "Find:" label); this causes them to stretch proportionally to the stretching of the dialog, as determined by the first rule of stretching. For example, imagine an attachment point for a vertical interactor edge positioned so that the point divides the horizontal distance between the left and right dialog edges in the ratio 1:2. After stretching, the attachment point will still divide the new distance in the same ratio.

The attachment points can be dragged away from the edges, in which case a line connects them visually to their respective edge. The meaning of the relation between an attachment point and its interactor edge is given by the second rule of stretching, below. In the other two selected interactors in Figure 6, all the attachment points have been dragged to the edges of the dialog (this is a bit hard to see for the "Find Next" button because the attachment points and their connecting lines are xor-ed over the picture). The common situation of positioning an attachment point at a dialog edge can be achieved by double-clicking the attachment point: this action will project it to the closest dialog edge (of the proper kind, horizontal or vertical); double clicking an interactor projects each of its attachment points to the closest dialog edge (of the proper kind).

> *Second rule of stretching: as a dialog stretches from a minimum size, the distance between an attachment point and its interactor edge remains constant.*

To satisfy the second rule of stretching, interactors may have to change their size under stretching. For example, all four edges of the central interactor in Figure 6 are attached to the corresponding dialog edges (the attachment points blend visually with the dialog border). Since the distance between an interactor edge and the corresponding dialog edge must remain constant under stretching, the interactor will have to become larger when the dialog stretches. This is the common situation in which we want one particular interactor to become "as large as possible" under stretching.

Another common situation is when we want an interactor to "stick to a corner". This happens for the "Find Next" button: its vertical edges are both attached to the east edge of the dialog (hence its horizontal size remains constant). Similarly its horizontal edges are both attached to the top (this is a bit hard to see in Figure 6) (hence, its vertical size must remain constant). Hence, all edges must maintain their distance from the north-east corner of the dialog.

What should we do about the box next to the "Find:" label? We probably want it to "stay to the top" of the dialog, hence we should attach its north and south edges to the north edge of the dialog. We also probably want it to stretch horizontally. Hence, we should attach its west edge to the west edge of the dialog, and its east edge to the east edge of the dialog. Note that the gap between it and the central interactor will remain constant.

An effective heuristic for setting a roughly correct stretching behaviour is to select all the interactors in a dialog and double click one of them. This will project all the attachment points of all the interactors to their corresponding closest dialog edge. (In the case of the card file dialog, this is exactly what we want: interactors close to the corners will stick to the corners, and interactors near the center will stretch in the appropriate direction.) The few incorrect attachments can then be fixed by hand.

So far we have considered attaching interactor edges to dialog edges only. This really covers the majority of simple situations, as in the case of the card file dialog. Finer control over stretching can be achieved by attaching interactor edges to arbitrary points within the dialog. As an excercise, imagine positioning the attachment points of two interactors lying next to each other horizontally, so that when the dialog stretches they each stretch by half the stretching amount, and maintain their distance to the edges of the dialog and between them.

There is no check against bad stretching behavior. If the attachment points are not set properly, some interactors may end up unintentionally overlapping each other. It is even possible to make interactors shrink as the dialog stretches. These anomalous situations can be easily detected by using the reshape control in the right corner of the header (while in stretching mode) to stretch the dialog and observe its stretching behavior. This process does not affect the minimum size of the dialog, which will be reinstated when switching back to edit mode.

3. Behavior

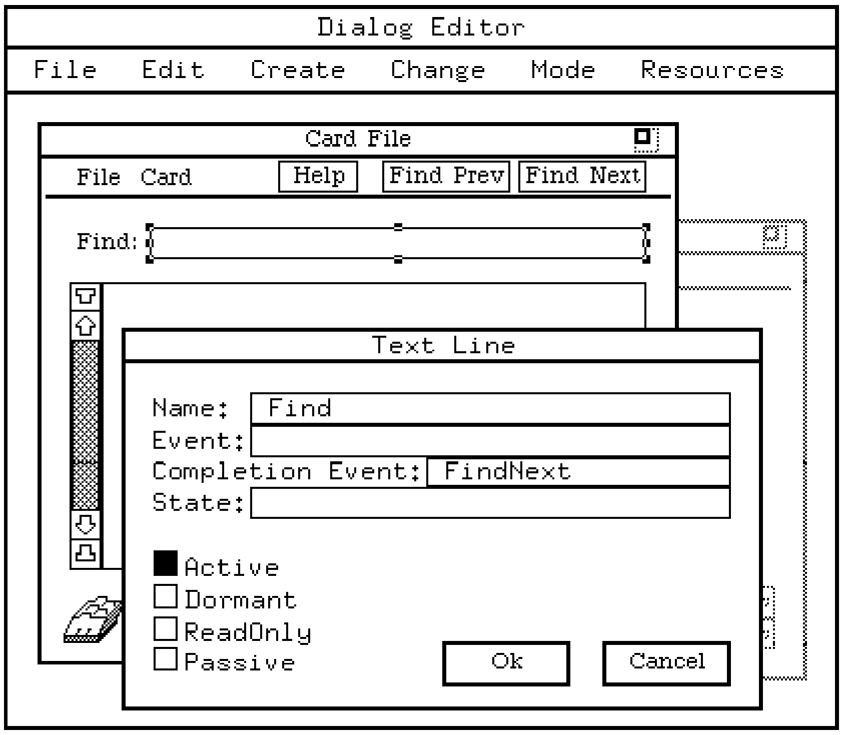
In this section we discuss the behavioral aspects (events, states and attributes) of user interface components. We also see how to test and debug the behavioral part of interfaces, and how to express some semantic relations (called groupings) between unconnected geometric entities.

3.1 Interactor attributes

Various classes of interactors are provided by the dialog editor; they include passive areas, buttons, menus, scroll bars, text areas and text lines (a complete list appears in the Appendix). For each interactor class there is an associated dialog used to set the attributes of interactors of that class. Figure 7 shows the attributes of the selected *text line* interactor; we now examine these attributes in turn.

Each interactor in a dialog may optionally be given a "Name" (so a program can refer to it), and the text line interactor in Figure 7 has been named "Find". Interactor names are forced to be unique within each dialog.

Most interactors generate *events* in response to user actions; events are represented as text strings. The text line dialog in Figure 7 does not specify an "Event", which would otherwise be generated whenever the text line contents are modified, but specifies instead a "Completion Event" which is generated when a carriage return is typed to the text line during normal operation. In this case the action corresponding to a completion event is to find the next occurrence of a string, hence the event is named "FindNext".



Find:

Name: Find
Event:
Completion Event: FindNext
State:

■ Active
☐ Dormant
☐ ReadOnly
☐ Passive

Figure 7: Interactor attributes

The text inside each text line forms its "State" attribute: in the figure this is currently empty in our text line. Most interactors have a "State" component, which by convention is always a text string representing the most important property of the interactor.

Finally, a text line can be in one of four states of activation. *Active* means that it functions normally. *Dormant* means that it does not function and looks dim: this is to warn the user that a text line is available but not currently enabled. *ReadOnly* means that the text line contents can be copied but not modified. *Passive* means that the text line looks normal but its contents cannot be copied of modified.

Other interactors in the card file dialog generate events, which are inspected and set by opening their respective attributes dialogs. The pulldown menus generate events for each menu selection. Buttons generate events when pressed. The central text region generates "Modified" events so the file card program can keep track of cards which have been modified. The scroll bar generates scroll events.

This notion of events provides a level of abstraction between the application and the user interface. The application receives high-level events, and does not have to deal with low-level mouse and keyboard actions. More interesting, many different low-level actions may generate the same high-level event. For example, the "Find Next" button in Figure 7 generates a "FindNext" event, just like the text line does. The client program receives a "FindNext" event from the dialog, but does not know whether it comes from a text line, a button, a menu, etc. Hence, the aspect of the user interface generating "FindNext" requests can be modified at will without affecting the client program.

3.2 Testing and debugging

It is very useful to try out a dialog without waiting until it is embedded in an application, in order to see how it looks "live". When the dialog editor is put in *test mode*, dialogs behave to some extent as they will behave in the application: menus pull down, buttons click, text can be written into text areas, etc.

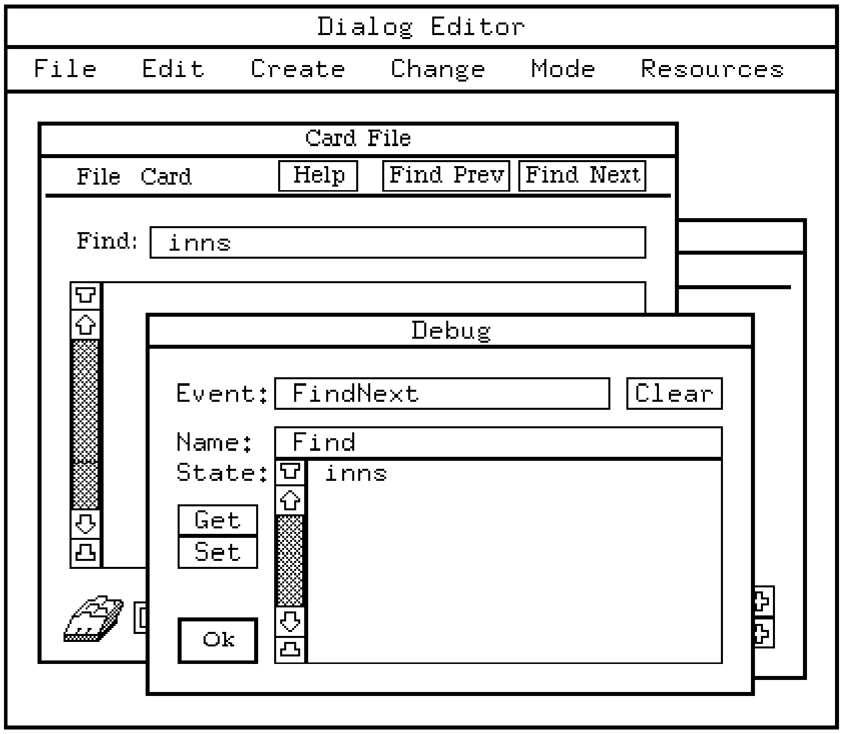


Figure 8: Debug dialog

Of course no semantics is attached to these actions, and dynamic behavior which happens under direct program control will not be performed. However, this limited testing has proven to be quite useful in practice for tuning the geometry and appearance of the dialog.

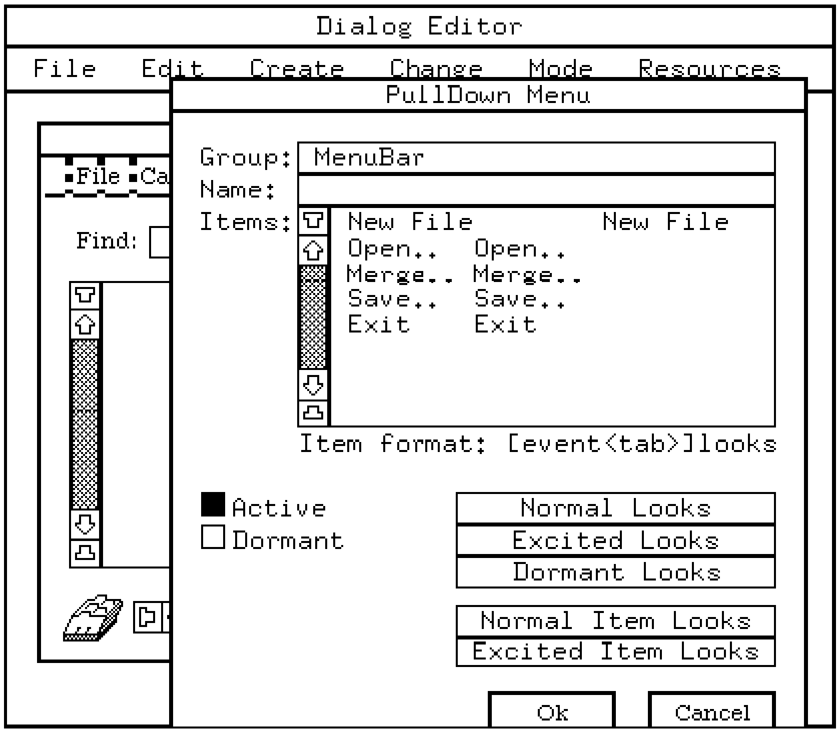
A further step is to put the dialog editor in *debug mode*. This is like test mode, but in addition a *debug dialog* appears which can be used to inspect the state of interactors and monitor the events they generate (often catching spelling mistakes in the attribute values).

Figure 8 shows the debug dialog after some interaction. We first typed "inns" followed by carriage return in the text line interactor next to the "Find:" label. This caused the event "FindNext" to be generated and to appear in the debug dialog. We then typed "Find" (the name of the text line) in the debug dialog and pressed the "Get" button to display the current state (contents) of the text line. We could also change the "inns" string in the debug dialog and press "Set" to set the new contents of the text line.

Using the debug dialog we can emulate the main information flow between a user interface and its client program: the program receives events in response to user actions, and it can then inspect and modify the dialog. (Client programs can also modify other attributes beyond the "State" attributes, but for simplicity this is not supported by the debug dialog.)

3.3 Interactor groups

Some collections of interactors function as logical units. These collections are called *interactor groups*, and are best illustrated by examples.



Group: MenuBar
Name:
Items:
```
New File        New File
Open..   Open..
Merge..  Merge..
Save..   Save..
Exit     Exit
```
Item format: [event<tab>]looks

■ Active
□ Dormant

Figure 9: Interactor groups

As a first example, a *radio button group* is a group of independently operated on-off buttons. Only one button in such a group can be in the "on" state at any given time; if another button is switched on, the previous one is set to "off". The dialog editor can group individual on-off buttons into radio button groups and can ungroup them.

Pulldown menus are another example of groupable interactors. When a menu in a group is pulled down by a mouse click, all the other menus in the same group can be pulled down by dragging the mouse over them, without having to release and click again. Figure 9 shows the attributes dialog of the "File" pulldown menu, which belongs to the "MenuBar" group, together with the "Cards" menu. Again, the dialog editor can group individual pulldown menus and can ungroup them.

Whenever a collection of similar groupable interactors (e.g. all on-off buttons or all pulldown menus) is selected, the operations of "grouping" and "ungrouping" can be invoked from the dialog editor. The meaning of these operations varies according to the kind of interactors being grouped.

The notion of groupable interactors is supported at an abstract level. The dialog editor can be extended with new classes of groupable interactors. When a new class of interactors is introduced, grouping and ungrouping operations can be specified for that class. If such operations are present, the dialog editor knows that such interactors can be grouped, and that the grouping operations should be invoked when appropriate. Details about how new interactor classes are introduced appear in a later section.

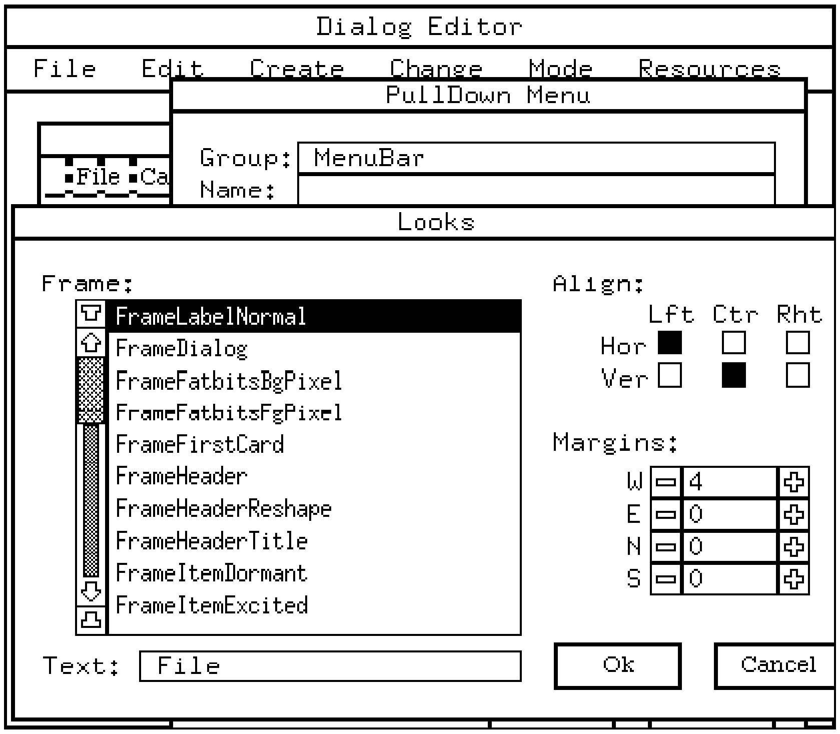
Incidentally, Figure 9 shows that pulldown menu items are specified just by listing them. They are duplicated because in some situations one may want to distinguish between what appears in the menu and the event which is generated when that menu item is selected.

4. Appearance

In this section we discuss the visual aspects (colors, fonts, borders, etc.) of user interface components, and show how the visual aspects of related components can be uniformly changed.

4.1 Looks

The appearance of interactors can be specified by setting their *looks*. Most interactors require more than one *look*, since their appearance changes depending on their state of activation. A single *look* is a fairly complex data structure, designed to achieve a compromise between generality and convenience.



Group: MenuBar
Name:

Frame: FrameLabelNormal

Align:

	Lft	Ctr	Rht
Hor	■	☐	☐
Ver	☐	■	☐

Margins:

W	4
E	0
N	0
S	0

Text: File

Figure 10: Interactor looks

One of the most important properties of looks is that they do *not* contain size information; looks are independent of geometry, and must adapt to the geometry of interactors. For example, a look may specify color and thickness of a border, but the dimensions of the border must change as the corresponding interactor stretches.

Figure 9 showed some buttons which can be used to set the looks of pulldown menu buttons and items. There are three looks for a pulldown menu button, one each for the normal state, the excited state (when the menu is pulled down), and the dormant state (when the menu is inactive). If we press the "Normal Looks" button, we get Figure 10.

Here we see that the normal look of the "File" button is a text look containing the string "File", which is left-aligned horizontally (with a west margin of four pixels), and vertically centered. The remaining properties of this look (font and color) are further specified by a *frame* called FrameLabelNormal.

4.2 Frames

In general, looks are composed of a *frame* and an *image* to be shown in the frame. The image can be either a uniform tint, a texture, a bitmap, or text in some font.

Just as a physical picture frame has a back board, a border and a piece of glass, so a frame has a *background*, a *border* and an *overlay* (the latter is used to achieve special translucence effects). Figure 11 shows a dialog open on the "FrameLabelNormal" frame: this is a font frame using the "FontBuiltIn" font. It has no border (its thickness is 0), but if it had one it would be painted in the "TintBd" tint.

Let us examine the components of frames in more detail.

A *tint* is a function mapping colors to colors, which is applied to the background during painting. Only some such functions are supported; basically, there are three classes: (a) *constant* tints opaquely paint a given color over the background (black or white for monochrome, RGB for color); (b) *swap* tints swap two given colors and have unpredictable effects on other colors (in monochrome, the unique swap tint inverts black and white); (c) the *transparent* tint leaves the background unaffected. Color tints are specified by RGB values (24 bits), and painted in the "closest" color found in the color map (8 bits). Color tints can also be set to "high quality", in which case the given RGB value is approximated by four color-map colors painted in 2x2 squares. Figure 12 shows the dialog used to set tint properties; the large area at the bottom displays the current color.

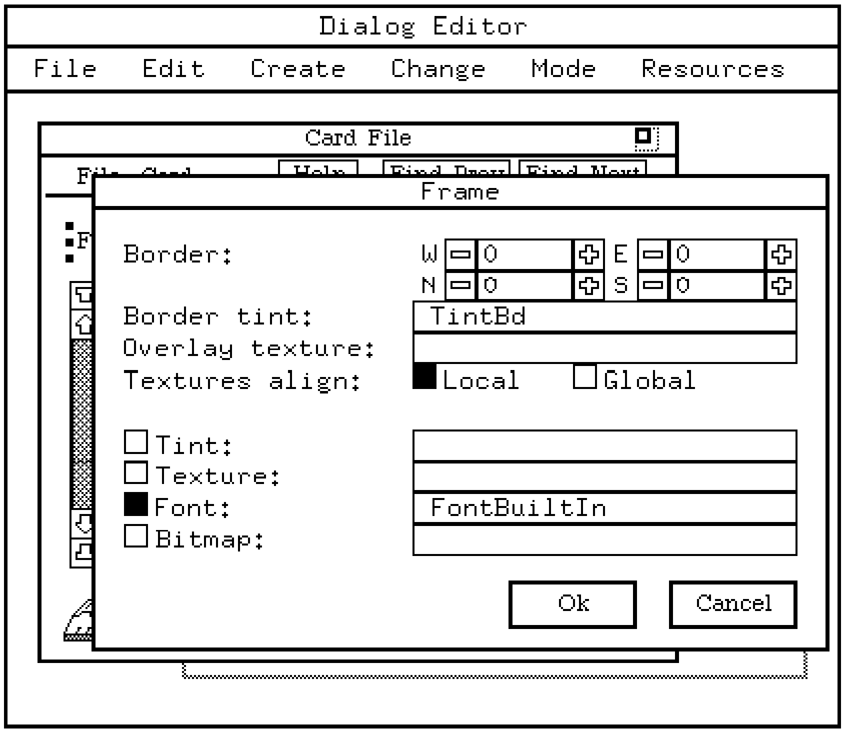
A *tint pair* is a pair of tints, used to paint black and white images in color.

A *texture* consists of two 16x16 bit patterns (normally also called textures), one for painting in monochrome and one for painting in color, plus a tint pair to specify the coloring (background tint for the "white" pixels, and foreground tint for the "black" pixels of the pattern).

A *bitmap* consists of an arbitrary-size bit pattern with a tintpair for painting. Optionally, one can specify an additional bit pattern of the same size (the *mask*) and an additional tint pair. This option allows one to paint bitmaps in three colors, normally one color each for the background, the borders and the inside of an image.

A *font* consists of a screen font and a tint pair for coloring.

A *frame* (Figure 11) consists of (a) border thicknesses for the four edges, (b) a border tint, (c) an overlay texture, and (d) an image which is either a tint, a texture, a font or a bitmap.



Border: W 0 E 0 N 0 S 0
Border tint: TintBd
Overlay texture:
Textures align: [✓ Local] Global
☐ Tint:
☐ Texture:
■ Font: FontBuiltIn
☐ Bitmap:

Figure 11: Frames

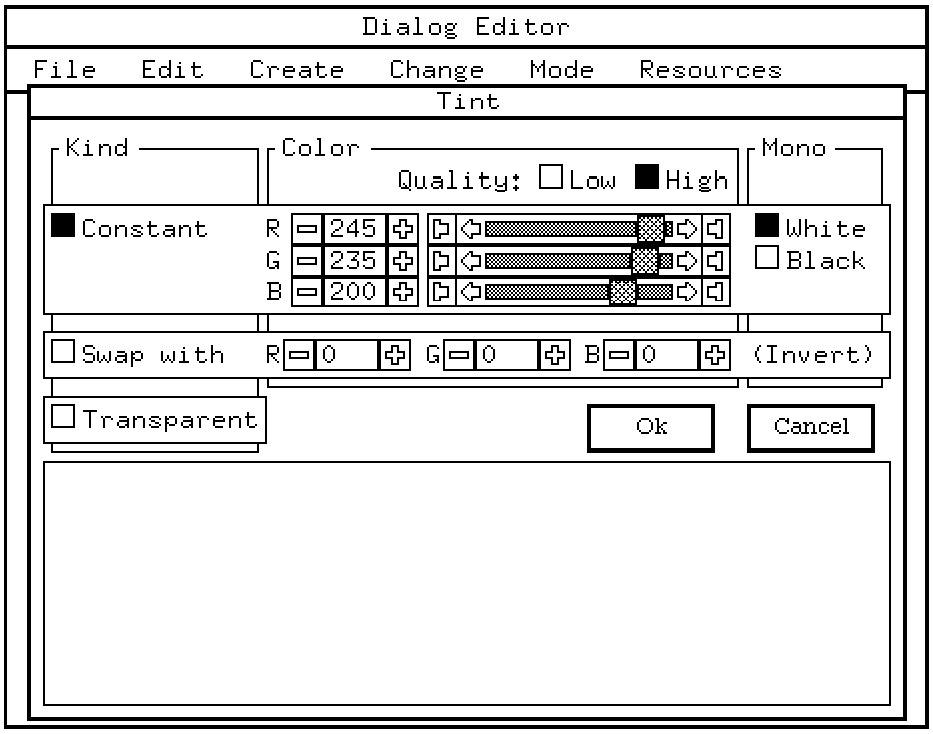
Finally, a *look* (Figure 10) specifies: (a) a frame; (b) a piece of text for font frames; (c) horizontal and vertical alignments and margins for font and bitmap frames. The alignments are left, center and right, horizontally, and top, middle and bottom, vertically. Margins determine distances between the edge of the looks, and the corresponding edge of non-centered text or bitmaps.

4.3 Dialog resources

While the looks of an interactor are local to it, the tints, tintpairs, textures, bitmaps, fonts and frames which define such looks are shared between all the interactors of a dialog, and are called *dialog resources.*

The sharing of dialog resources allows one to change the looks of many interactors at once. For example, imagine wanting to change the background color of a dialog. The background is composed of the actual background of the dialog, but also of the background of many interactors (e.g. borderless labels). It would be very painful to have to select each interactor in turn, and change the appropriate looks. Similarly, imagine wanting to change the font of all the interactors, or maybe the font of all text interactors but not the font used in labels.

The solution to these global replacement problems is provided by changing the dialog resources, as opposed to changing individual interactors. Changing a resource affects the looks of all the interactors which refer to it. Each resource is identified by a resource name (a string) within another resource or a look. All resources point to each other through symbolic names. For example, a tint pair is composed of the *names* of two tints. A menu in the dialog editor gives access to the list of all resources in a dialog.



Quality: ☐ Low ■ High

■ Constant R 245 G 235 B 200 ■ White ☐ Black

☐ Swap with R 0 G 0 B 0 (Invert)

☐ Transparent

Figure 12: Tints

Here is an example of how to modify resources. A look L may mention a frame called "FrameDefault", shared by many other looks. To change the appearance of L, one can replace the string "FrameDefault" by "FrameSpecial". If "FrameSpecial" already exists in the list of dialog resources, the looks L will change accordingly, otherwise a new "blank" frame will be created which can then be further defined. To change the appearance of all the looks which mention "FrameDefault" one selects "FrameDefault" from the resources menus, and modifies it (which involves modifying more resource names).

Newly created dialogs have a predefined collection of resources, which are used for the default looks of the available interactors. Unused dialog resources are garbage collected when saving to disk. The standard resources are made available again when loading a dialog into the dialog editor (unless there are already different resources with the same name)

Groups of resources can be copied and pasted across dialogs.

5. Meaning

In this section we discuss how to give meaning to a user interface by specifying its connections to the application.

5.1 Attaching procedures to events

Meaning is given to a user interface by attaching procedures to interactor events. As we have already explained, high-level events are produced by interactors in response to low-level user actions.

The attachment is achieved by *registering* procedures with a dialog to respond to given events; when such events are generated, the corresponding procedures are invoked directly. Hence the client program does not have to poll the dialog, nor is there a main interaction loop to be programmed: the client program simply registers event routines and then passes control to the dialog, which takes care of the flow of control. This is sometimes called *event-driven* programming.

To illustrate how to attach procedures to events, we present a simple but complete application whose code is less than two pages of Modula2+ [Rovner 85]. The application is a file editor which can load, modify and save a single text file at a time. Figure 3 showed the user interface of this application, which consists of a scrollable text area and a pulldown menu.

Here is the code of the application:

```
SAFE MODULE FileApplication;      (* A simple single-file text editor. *)
IMPORT Time, Text, Rd, Wr, FileStream, RootDialog, InitInteractors, FileDialogVBT;

TYPE
  Argument = REF RECORD
    dialog: RootDialog.T;
    fileDialog: FileDialogVBT.T;
  END;

PROCEDURE ReadProc(argument: REFANY; fileName: Text.T);
  VAR arg: Argument; text: Text.T; rd: Rd.T;
  BEGIN
    arg := NARROW(argument, Argument);
    rd := FileStream.OpenRead(NIL, fileName);
    text := Rd.GetText(rd, Rd.Length(rd));
    Rd.Close(rd);
    RootDialog.SetTextProperty(arg^.dialog, "File", "State", text, TRUE);
  END ReadProc;

PROCEDURE WriteProc(argument: REFANY; fileName: Text.T);
  VAR arg: Argument; text: Text.T; wr: Wr.T;
  BEGIN
    arg := NARROW(argument, Argument);
    text := RootDialog.GetTextProperty(arg^.dialog, "File", "State");
    wr := FileStream.OpenWrite(NIL, fileName);
    Wr.PutText(wr, text);
    Wr.Close(wr);
  END WriteProc;

PROCEDURE NewProc(argument: REFANY; dialog: RootDialog.T; eventName: Text.T;
    eventValue: REFANY; eventTime: Time.Ticks);
  VAR arg: Argument;
  BEGIN
    arg := NARROW(argument, Argument);
    RootDialog.SetTextProperty(arg^.dialog, "File", "State", NIL, TRUE);
  END NewProc;

PROCEDURE OpenProc(argument: REFANY; dialog: RootDialog.T; eventName: Text.T;
    eventValue: REFANY; eventTime: Time.Ticks);
  VAR arg: Argument;
  BEGIN
    arg := NARROW(argument, Argument);
    FileDialogVBT.Interact(arg^.fileDialog, ReadProc, arg, eventTime);
  END OpenProc;

PROCEDURE SaveProc(argument: REFANY; dialog: RootDialog.T; eventName: Text.T;
    eventValue: REFANY; eventTime: Time.Ticks);
  VAR arg: Argument;
  BEGIN
    arg := NARROW(argument, Argument);
    FileDialogVBT.Interact(arg^.fileDialog, WriteProc, arg, eventTime);
  END SaveProc;

PROCEDURE ExitProc(argument: REFANY; dialog: RootDialog.T; eventName: Text.T;
    eventValue: REFANY; eventTime: Time.Ticks);
  BEGIN
    RootDialog.Terminate(dialog, eventTime);
  END ExitProc;

PROCEDURE Run;
  VAR arg: Argument;
  BEGIN
    NEW(arg);
    arg^.fileDialog := FileDialogVBT.New();
    arg^.dialog := RootDialog.New("File Application", "Icon", "Dialog");
    RootDialog.Register(arg^.dialog, "New", NewProc, arg);
    RootDialog.Register(arg^.dialog, "Open..", OpenProc, arg);
    RootDialog.Register(arg^.dialog, "Save..", SaveProc, arg);
    RootDialog.Register(arg^.dialog, "Exit", ExitProc, arg);
    RootDialog.Interact(arg^.dialog);
  END Run;

BEGIN
  RootDialog.Init;
  InitInteractors.Init;
  FileDialogVBT.Init;
  Run;
END FileApplication.
```

First, starting at the bottom, some initialization routines are called. `FileDialogVBT` is a module providing a pop-up dialog for reading and writing files (this dialog is visible in Figure 4).

The `Run` procedure is then invoked; it creates a new `FileDialogVBT` object and a new `RootDialog` object (i.e. a top-level dialog for an application). The file containing the application's dialog, here called `"Dialog"`, is found by following a directory path which is specified elsewhere.

Once we have a dialog, we can start registering procedures to respond to the events the dialog generates: these are the events produced by the pulldown menu. (The last parameter to `RootDialog.Register` is an argument record which is passed back to the event procedure when this is invoked.) Finally, a call to `RootDialog.Interact` gets things started: the application window is installed into the window system and interaction with the user may begin.

The simplest event routine is `ExitProc`; it simply calls `RootDialog.Terminate` to close the application window and exit. All the event procedures must have the same type, and hence the same parameters; many of these parameters may remain unused in most situations, as is the case in `ExitProc`. The Modula2+ REFANY type and the NARROW construct represent dynamic types and dynamic typechecking respectively.

5.2 Accessing the dialog state

When event procedures are called, they must be able to examine the dialog in order to read parameters or set results.

Let us see how this works in `NewProc`, whose purpose is to clear the file being edited and revert to an empty file. This procedure consists of a call to `RootDialog.SetTextProperty`, whose parameters are: (a) the dialog; (b) the name of an interactor within the dialog (`"File"` is the scrollable text area); (c) the name of a property of that interactor (`"State"` denotes the contents of the text area); (d) the new value to be set for that property (`NIL` will set it to the empty text); and (e) a repaint flag.

Here we see that interactors within a dialog are accessed by symbolic names, so that programs do not have to know what kind of interactors they are dealing with. Different interactors having the same name could be substituted without affecting the program, provided they have a similar event protocol and properties.

When a program refers to a property of an interactor, the program relies on knowing that a given property makes sense for a given interactor. In such a case, we have a dependency on the inner structure of the dialog, which in general should be avoided. However, this dependency is usually very weak because many interactors have similar properties, and even if we know the name of the appropriate property it might not say much about the interactor itself. In the example above, "State" is a property of almost all interactors, and certainly of all text interactors.

The `SaveProc` is not very different from the `NewProc`, but shows the use of pop-up file dialogs. A call to `FileDialogVBT.Interact` causes a dialog to pop up so the user can type a file name. The procedure passed to `FileDialogVBT.Interact` (`WriteProc`, in this case) is invoked with the file name typed by the user. `WriteProc` extracts the current contents of the text area, and attempts to write them to the file. If `WriteProc` fails (e.g. because the file name is illegal), `FileDialogVBT.Interact` intercepts the failure and presents an error message to the user, who can try again with a different file name. If `WriteProc` is successful, the file dialog is taken away.

`OpenProc` and `ReadProc` are similar to `SaveProc` and `WriteProc`. This completes the description of the application.

6. Architecture

This is a bottom-up review of interactors, dialogs and the architecture of the dialog editor. Many of the features described here have already been encountered, but are repeated for completeness.

6.1 Windows

Interactors and dialogs are implemented on top of the Trestle window system [Manasse 87], which provides a window abstraction called VBT (for Virtual Bitmap Terminal). Trestle supports hierarchies of VBT's, where a VBT can be a *leaf* (a proper window) or a *split* composed of other VBT's. A split can organize its children into overlapping or tiling arrangements (or other less common arrangements). A dialog is a tiling split composed of a header and a body, and the body is an overlapping split of interactors. Each interactor is a VBT, and some interactors are themselves splits.

Interactors and dialogs are "real" windows: they could be directly inserted into the window system. Dialog-based applications are very window-intensive: it is not uncommon for them to use hundreds of VBT's.

6.2 Interactors

An interactor is a "user interface primitive"; something which is seen as a functional unit. It can be used to present some internal state, to operate a mechanism, or to modify a structure. Normally, an interactor has a well defined geometrical extent.

Some interactors are composed of other interactors (several have embedded scroll bars, which are also interactors). Such complexity in something which is supposed to be a primitive is admissible when there is a very tight semantic coupling between the embedded interactors. Whenever possible, composite interactors should be avoided.

Interactors have a given "reactivity" at any point in time, that is they can be functioning at various levels of activation or sleepiness. Here are the standard meanings for the reactivities currently in use:

Active:	Functions normally.
ReadOnly:	Functions but cannot be modified (to some degree).
Protected:	Needs special action to become temporarily active.
Passive:	Does not function, but looks normal.
Dormant:	Does not function and looks "dim".

Interactors have one or more *looks* specifying their appearance. There is a common looks data structure used by all interactors.

Interactors have two interfaces, in a sense. One is a standard Modula2+ interface, providing routines to create interactors, activate them, set and inspect their properties and appearance. This interface is suitable for clients who want to use interactors in isolation, independently of dialogs, and is not described here in any detail. The second interface is intended for interactors which are embedded in dialogs. It is based on multiple-inheritance subclassing, which is implemented in (but not directly supported by) Modula2+, and which is described below and in the Appendix.

All interactors must be subclasses of the "Interactor" class, which is defined as any VBT providing the following four methods: (1) GetProperty: get the current value of a property of an interactor; (2) SetProperty: set the value of a property of an interactor; (3) GetDescription: produce a standard data structure which completely describes an interactor; (4) SetDescription: make an interactor conform to a given description. Moreover, two routines for creating and deleting interactors of a given class must be provided; such routines have the same type for all interactor classes.

The data structure returned by GetDescription (called a *description*) is suitable to be stored on disk; it is a list of name-value pairs, where the name is a Text.T (an immutable character string) and the value is a REFANY (an arbitrary pointer, dynamically typechecked).

To implement a new interactor class, one can take any existing VBT performing some function and supply the four methods and two routines above. This will allow the new interactor to be inserted in dialog boxes and to be manipulated by the dialog editor.

We have claimed in the introduction that the specialized VBT's to be used in a single application should be made into interactors in order to be inserted into dialogs. However, the task of making an interactor out of a one-off VBT can be distracting, hence a shortcut is provided. A special class of *generic* interactors is provided: they look like grey rectangles in the dialog editor, and have a single property ("VBT") that can be assigned only at run-time. At run time, these generic interactors can be given an arbitrary VBT, and they "become" that VBT. From then on, this foreign VBT appears in the dialog, but is handled by bypassing the dialog abstraction (i.e. there is no notion of registering event procedures for generic interactors).

This violation of abstraction turns out to be very useful for experimentation, since application builders can insert fragments of applications into dialogs with no overhead (even applications which had been built independently of dialogs). For example, a specialized graphical editor can be implemented as a VBT using whatever interaction style is considered necessary; this specialized editor is then embedded into a standard dialog as a generic VBT, taking advantage of the standard menus, buttons, etc.

6.3 Dialogs

A dialog is a "user interface subroutine". It is, visually, just a collection of interactors, but from the client point of view each dialog consists of a list of named interactors, and an association of client procedures with abstract "events" (text strings) generated by the interactors.

Interactors in a dialog are set up to generate events in response to user actions; the dialog receives those events and calls the corresponding client procedures. The client procedures can then inspect and modify the state of the dialog and its interactors.

The state of a dialog consists mostly of the state of its interactors, and this is accessible only by knowing the name of a particular interactor and operating on it by "GetProperty" and "SetProperty" methods.

Dialogs also support a notion of "interactor groups", as we have seen. Each interactor is optionally associated with a "group" interactor (there are no groups of groups).

Dialogs have a minimum size; when placed in a window smaller than their minimum size they are incompletely shown. Dialog elements have stretching information, which determines what happens to them when the dialog changes sizes beyond its minimum size.

Dialogs are saved to disk as data structures (in fact, as *descriptions*, as returned by the GetDescription routines). To use them, client programs have to (a) load them from disk, (b) register procedures to respond to events, and (c) pop them up and take them down.

Dialogs have no knowledge of what kind of interactors they contain; they manipulate their interactors through generic procedures which can be applied to all interactors. One might think that the module implementing dialogs imports all the interactor modules, but actually things work the other way around, to maintain dialogs independent of any given set of interactors. Each interactor module imports the dialog module, and registers creation and deletion routines for that interactor with the dialog module. These routines are inserted in an association list, together with the kind of interactor they apply to (as a string).

The dialog module must be able to read a dialog description from disk and convert it into a dialog; this implies creating the necessary interactors. The dialog description contains the kind of each interactor as a string (e.g. "PullDownMenu"). The dialog module then searches the list of registered interactor creation routines; if it finds a routine under that kind, it is able to create the corresponding interactor, otherwise a run-time error is reported.

6.4 Dialog editor

The Modula2+ dialog interface provides routines which can be used to build dialogs and user interfaces by brute force programming, although this is discouraged. These routines are used by the dialog editor for its normal operation, and can also be used by client programs, for example to dynamically change the number or properties of interactors in a dialog, to move them around, or to hide and expose them.

The dialog editor is structured so that it does not have any knowledge of the kind of interactors it is dealing with, but still is able to change interactor properties and looks. This is achieved as follows. When a new interactor class (I) is added to the dialog editor, a new *matching* dialog (D) must be supplied with it. This dialog has one field for each property of the interactor (i.e. for each property P of I, D has has one interactor whose name in the dialog is P, and whose "State" property represents the value of P).

When the user selects "Change Attributes" in the dialog editor, the interactor is asked to supply a *description* of itself (a list of <name,value> pairs). This description is transformed, in a uniform way, into a *dialog setting* (which is also a description consisting of <name, <"State", value>> triples). The dialog is then told to assume this setting and is popped up, hence showing the current state of the interactor. The user can then modify the dialog; when the changes are done the dialog is asked to supply its dialog setting, which is converted back into a description. The description is then forced upon the interactor, which assumes the new state. In all this process, the dialog editor has no idea of what the interactor does, what the dialog contains, or what the properties mean.

The looks of an interactor are just some of its properties, hence they can be modified as described above (The "matching" dialog of an interactor may contain some special buttons, called "EditLooks" buttons, which pop up dialogs for defining looks properties.)

Since the dialog editor itself uses dialogs in its normal operation, there is a small problem of bootstrapping the interface. The first version of the editor used dialogs built "by hand" which were quickly replaced by editor-generated dialogs as soon as there was sufficient critical mass.

7. Conclusions

7.1 Discussion

We have described a set of modules and tools for building certain classes of user interfaces. The general approach is to provide a set of ready-to-use interaction primitives, generically called *interactors* (buttons, menus, text areas, etc.) and to use an editor to assemble them into user interfaces.

One way of assembling basic interactors into more complex objects can be provided at the window system level through the notion of sub-window managers (called *splits* in Trestle [Manasse 87], and *geometry managers* in the X window system [Scheifler 86]);

each window can be split into sub-windows, each containing other splits or windows (e.g. interactors). We have described a way of composing interactors by the notion of *dialogs*; dialogs are specialized sub-window managers, which in addition present a standard simplified event interface to client programs.

To put this approach into context, let us consider alternatives. The obvious one is to let users build their own interfaces from scratch, using bare window system interfaces. This has many disadvantages, both in terms of discouraging people from building user interfaces, wasting people's time, generating bad interfaces and producing inconsistent interaction styles. Probably, some of this activity will always be required, but the hope is that it will be limited to generating specialized interactors when really needed, and that such interactors will be combined with the standard ones whenever possible within a dialog (or similar) paradigm.

The next step up is to provide a set of tools to help application programmers to build user interfaces. This has produced various *toolkits* and *user interface management systems* at various levels of sophistication [Pfaff 83, Schulert 85]. These are libraries which provide a set of standard components and composition methods, and a programming policy of how to use them.

One can even go to the extreme of setting up an environment where totally standardized user interfaces are generated with little or no user control over the geometry of the interface. This meets the abstraction and quick-turnaround criteria, but leaves much to be desired in flexibility and aesthetics.

Instead of supplying a fixed set of interactors, one could try to provide a special-purpose language for specifying the appearance and behavior of interactors [Cardelli 85, Green 86] (window interfaces are a low-level example of this). Such a language gives more flexibility and coherence in building libraries of interactors, but is very hard to provide in a clean and effective way; note that ordinary sequential languages are ill suited, because of the complex non-deterministic and concurrent activities present in user interfaces. The approach taken in this work is instead to provide a fixed set of basic interactors known to be useful for most interfaces, and to structure the environment so that new interactors can be easily added (by making sure that dialogs do not care, or know, what kind of interactors they contain).

Another approach is to supply a graphical *interactor editor* for building certain classes of interactors, instead of building them by a special purpose or a general purpose language. This is similar to the dialog editor approach, but one level lower, and has been attempted with some success [Myers 86]. This idea runs into some difficulties, on one side, by its inability to build arbitrary interactors, and on the other side, by the existence of relatively small sets of interactors which can cover most standard situations.

Finally, one could design a "user interface specification language" to specify geometry, appearance, behaviour, etc. [Jacob 86]. This is extremely ambitious and maybe too hard. Textual languages tend to be very inconvenient when one is trying to specify two-dimensional information. Hence the message is that user interfaces should not be described by languages, but should be built by direct manipulation.

7.2 Related work

The dialog paradigm comes from the Macintosh environment [Apple 85], with roots in work done at Xerox PARC. The Macintosh *Resource Editor* also includes a dialog editor. However, Macintosh dialogs do not provide any abstraction: dialog clients directly access the dialog components.

The stimulus for building a dialog editor came from a timely talk by Jean-Marie Hullot about a user interface editor in a Lisp environment, which later developed into the ExperTelligence Interface Builder [Hullot 86, Hullot 87]. The latter program turned out to be very similar to this work, modulo the diversity of the underlying environments.

7.3 Environment and acknowledgements

The software described in this paper is written in Modula2+ [Rovner 85] and runs on Firefly personal multiprocessors [Thacker 87] under the Taos operating system [McJones 87] and the Trestle window system [Manasse 87]. The tint model comes directly from Trestle.

Greg Nelson and Mark Manasse provided the window system machinery necessary to make interactors and dialogs work smoothly.

The current stretching model, generalizing a previously implemented model, was devised in a discussion with Leslie Lamport.

Appendix: Supplied interactors

This appendix describes the kinds of interactors available at the time of writing this report, together with their properties. Neither the interactors nor their properties are critical to the general architecture. New interactor classes can be added to the dialog editor by adding a single program line and recompiling (in lack of dynamic linking). New properties can be added to interactors without changing any of the interfaces.

A.1 Passive Areas

A passive area constantly shows one *look*. Passive areas are mostly used to provide backgrounds, borders, text labels and bitmap icons. As interactors, they have the following *description* in terms of name-value pairs:

"Kind" = "PassiveArea"
"Reactivity" = "Active" or "Dormant"
"NormalLooks", "DormantLooks" : IconLooks.T

"Kind" is a property of all interactors (a constant string describing the kind of interactor), and IconLooks.T is the Modula2+ data type of looks.

A.2 Trill Buttons

A trill button generates events on mouse down-transitions, and then generates more events (by auto-repeating, up to the corresponding up-transition) while the mouse cursor is on top of the button.

"Kind" = "TrillButton"
"Event" : Text.T
"Reactivity" = "Active" or "Dormant"
"NormalLooks", "ExcitedLooks", "DormantLooks" : IconLooks.T

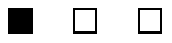
The "ExcitedLooks" are shown while the button is generating events.

A.3 Trigger Buttons

Trigger buttons are stateless buttons which generate events on mouse transitions. There are both down-triggered and up-triggered flavors. If "protected", one has to click twice to activate them.

"Kind" = "TriggerButton"
"Event" : Text.T
"Transition" = "Up" or "Down"
"Reactivity" = "Active", "Dormant" or "Protected"
"NormalLooks", "ExcitedLooks", "DormantLooks",
    "ProtectedLooks": IconLooks.T

A.4 Radio Buttons



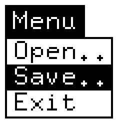
An individual radio button has one bit of state; when used in isolation it is also called an on/off button, since it can be put into an "on" or "off" state, and toggled by mouse clicks. Events are generated on state changes.

Radio buttons can be logically "grouped" while remaining physically separated. If there are many buttons in the same group, only one of them can be "on", (they can all be "off"). Clicking one button in a group will switch off any button that is "on" and set the current one to the "on" state.

An individual radio button is an interactor with the following description:

"Kind" = "RadioButton"
"Event" : Text.T **("something changed" event)**
"State" : Text.T **(either OffName (shared by the group)**
"StateOnName" : Text.T **or one of the group's OnNames)**
"StateOffName" : Text.T **(shared by the group)**
"Transition" = "Up" or "Down"
"Reactivity" = "Active", "Dormant" or "Protected"
"OffLooks", "ExcitedOffLooks", "OnLooks", "ExcitedOnLooks",
"DormantLooks", "ProtectedLooks" : IconLooks.T

A.5 Pulldown Menus



A pulldown menu is a button which pops up a list of selections when the mouse is pressed over it.

"Kind" = "PullDown"
"State" : Text.T
"Reactivity" = "Active" or "Dormant"
"NormalLooks", "ExcitedLooks", "DormantLooks" : IconLooks.T

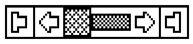
The following applies to all the menu items:

"NormalMenuLooks", "ExcitedMenuLooks": IconLooks.T.
(text looks)

where "State" is a list of menu items terminated by '\n', each with format: "event '\t' looks", where "event" is the event generated when that item is selected, and "looks" is the string which appears in the menu.

Pulldowns can be grouped. Menus in the same group can be pulled down by rolling the mouse over them, after the first down-click.

A.6 Scroll Bars

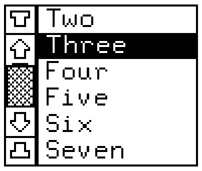


A scroll bar maintains four numeric quantities; a low and high bound, and a low and high range, in the relation lowBound ≤ lowRange ≤ highRange ≤ highBound. The relative size of the ranges with respect to the bounds determines the size of a visible cursor which the user can "thumb" along a bar. The user can also click on the bar on either side of the cursor to "page", click two buttons on either side of the bar to "scroll", and click two buttons on either side of the scroll buttons to move the cursor to the "top" and "bottom" of the admissible range. Although a standard interpretation has just been given for the admissible user actions, the events generated by those actions are uninterpreted, and can be defined to have different meanings.

"Kind" = "ScrollBar"
"ToLowEvent", "StepDownEvent", "SlideDownEvent", "ThumbEvent",
"SlideUpEvent", "StepUpEvent", "ToHighEvent": Text.T
"Reactivity" = "Active", "Dormant" or "Passive"
"Axis" = "Hor" or "Ver"
"LowBound", "LowRange", "HighRange", "HighBound": Text.T
(actually integers)
"ScrollBgLooks", "SliderSlotLooks", "SliderThumbLooks"
"ScrollJumpOffLooks", "ScrollJumpOnLooks",
"ScrollStepOffLooks", "ScrollStepOnLooks" : IconLooks.T

where the "Jump" and "Step" looks describe the bitmaps used for the "to low" and "step down" buttons in their north-pointing orientation.

A.7 Browsers



A browser is used to select from a variable-size (possibly large) set of items. Only a subset of the items is visible, and a scroll bar is used to access the rest of them. In "Single Selection" mode, items are selected by clicking and dragging; the up-transition determines the selected item. In "Multiple Selection" mode, items are toggled between the selected and unselected state by clicking on them or sweeping over them. An option-click "between" two items, inserts all the currently selected items between those items, rearranging the list. A browser has the following description:

"Kind" = "Browser"
"State" : Text.T
"SelectionMode" = "Single" or "Multiple"
"Reactivity" = "Active", "Dormant", "ReadOnly" or "Passive"
"TextLooks" plus the scrollbar looks: IconLooks.T;

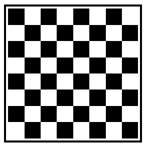
where "State is a list of menu items terminated by '\n', each with format: "['+' | '-'] looks '\t' upevent '\t' dnevent". If the first character is a '+' the item is selected; if it is a '-' the item is non-selected; if it is something else it is considered part of "looks" and the item is non-selected. "looks" is the string which appears in the browser, "upevent" is the event generated when that items is deselected, and "dnevent" is the event generated when that item is selected.

The following property is defined for the GetProperty method:

"Selection" : Text.T

this is a list of looks fields, separated by '\n', one for each currently selected item (at most one in single selection mode).

A.8 Fatbits



A fatbits interactor maintains a magnified display of bits. The bits themselves are not maintained by the interactor, so it can be used for textures, cursors, bitmaps, etc.

"Kind" = "Fatbits"	
"FatbitSize" : Text.T	**(a cardinal, in pixel units)**
"HorSize", "VerSize" : Text.T	**(cardinals, in fatbits units)**

"PressEvent", "DragEvent", "ReleaseEvent" : Text.T
"Reactivity" = "Active", "Dormant" or "Passive"
"BackgroundLooks", "BgTintLooks", "FgTintLooks" : IconLooks.T

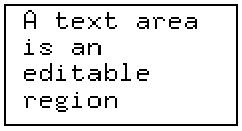
where "PressEvent" is generated when one fatbit is activated by a down click, "DragEvent" is generated when one fatbit is activated by dragging, and "ReleaseEvent" is generate on the up click. The coordinate of the fatbits is passed as the "eventValue" (a RefPoint) to the procedures registered for those events.

The following properties are defined for the SetProperty method:

"Bitmap" : RefBitmap	**(paint a fatbits from a bitmap, using the fg and bg tints)**
"Fg" : RefPoint	**(set a fatbits point to the foreground tint)**
"Bg" : RefPoint	**(set a fatbits point to the background tint)**
"Paint" = "Fg" or "Bg"	**(paint the entire fatbits in fg or bg tint)**

"Transform" = "TumbleW", "TumbleE", "TumbleN", "TumbleS",
"RotateCW", "RotateCCW", "FlipH" or "FlipV"
(perform a transformation)

A.9 Text Areas


A text area
is an
editable
region

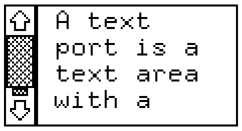
A text area is an editable region of text.

"Kind" = "TextArea"	
"Event" : Text.T	**("something changed" event)**
"State" : Text.T	**(the entire text in the text area)**
"Reactivity" = "Active", "Dormant", "ReadOnly" or "Passive"	
"TextLooks" : IconLooks.T	

The following properties are are recognized by SetProperty:

"FocusOn" : RefTime	**(grab the focus)**
"SelectAll" : RefTime	**(select the entire text)**

A.10 Text Ports

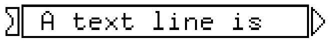

A text
port is a
text area
with a

A text port is a text area with a scroll bar.

"Kind" = "TextPort"	
"Event" : Text.T	**("something changed" event)**
"State" : Text.T	**(the entire text in the text port)**
"Reactivity" = "Active", "Dormant", "ReadOnly" or "Passive"	
the text looks plus the scrollbar looks : IconLooks.T	

SetProperty recognizes the text area properties.

A.11 Text Lines

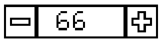

A text line is

A text line is a text area showing a single line of text. When the text overflows, buttons for horizontal scrolling become visible. A text line intercepts carriage returns so that they are not inserted into the text, and are interpreted as "completion events" (meaning "do it").

"Kind" = "TextLine"	
"Event" : Text.T	**("something changed" event)**
"CompletionEvent" : Text.T	**(on carriage return)**
"State" : Text.T	**(the entire text in the text line)**
"Reactivity" = "Active", "Dormant", "ReadOnly" or "Passive"	
"LftArrowHeadLooks", "LftArrowTailLooks" plus the text looks: IconLooks.T	

The "Arrow" looks are the bitmaps used for scroll buttons in their left-pointing versions. SetProperty recognizes the text area properties.

A.12 Numeric Areas


66

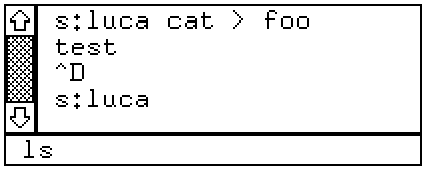
A numeric area is a text line maintaining a numeric quantity between two bounds. The numeric value can be modified by editing, or by clicking "increment" and "decrement" buttons. Legal strings are numeric strings, "infinity", "+infinity" and "-infinity". Unrecognized strings are ignored, and the numeric value is unchanged. The strings are parsed and checked for legality and in-boundness when typing carriage return, or when clicking the increment and decrement buttons.

"Kind" = "NumericArea"	
"Event" : Text.T	**("something changed" event)**
"CompletionEvent" : Text.T	**(on carriage return)**
"State" : Text.T	**(the integer)**
"LowBound", "HighBound" : Text.T	**(the integer bounds)**
"Reactivity" = "Active", "Dormant", "ReadOnly" or "Passive"	
"IncDecButtons" = "Yes" or "No"	**(increment/decrement butt. visible)**
"NumIncOffLooks", "NumIncOnLooks", "NumDecOffLooks",	

"NumDecOnLooks" plus the text line looks : IconLooks.T

SetProperty recognizes the text area properties.

A.13 Transcripts



```
s:luca cat > foo
test
^D
s:luca
```

```
ls
```

A transcript is a text line (input) and a text port (output) packaged to provide a typescript with editable history.

"Kind" = "Transcript"	
"Event" : Text.T	**(input area modified)**
"CompletionEvent" : Text.T	**(cr in the input area)**
"InState" : Text.T	**(the entire text in the input area)**
"OutState" : Text.T	**(the entire text in the output area)**
"Reactivity" = "Active", "Dormant", "ReadOnly" or "Passive"	
the text port and the text line looks: IconLooks.T	

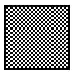
GetProperty recognizes the following property, extracting the contents of the input area and resetting it to empty.

"State" : Text.T

SetProperty recognizes the following property, appending a text to the end of the output area.

"State" : Text.T

A.14 Generic Interactors



A "generic" VBT interactor can host an arbitrary VBT. This way, arbitrary VBT's can be embedded into dialogs without having to make them into full interactors.

"Kind" = "Generic"

The following property is defined for SetProperty:

"VBT" : VBT.T	**(turn the generic vbt into a specific vbt)**

References

[Apple 83] Apple Computer Inc. **MacDraw**, program documentation.

[Apple 85] Apple Computer Inc. **Inside Macintosh**, Addison-Wesley, 1985.

[Buxton 80] W.A.S.Buxton, R.Sniderman: **Iteration in the design of the human-computer interface.** *Proc. of the 13th Annual Meeting, Human Factors Association of Canada.* 1980, pp. 72-81.

[Cardelli 85] L.Cardelli, R.Pike: **Squeak: a language for communicating with mice**, *Twelfth ACM Annual Conference on Computer Graphics and Interactive Techniques* (SIGGRAPH 85).

[Green 86] M.Green: **A survey of three dialogue models**, *ACM Transactions on Graphics*, Vol. 5, no. 3, July 1986, pp 244-275.

[Jacob 86] R.J.K.Jacob: **A specification language for direct-manipulation user interfaces**, *ACM Transactions on Graphics*, Vol 5, no 4, October 1986, pp. 283-317.

[Hullot 86] J-M.Hullot: **SOS Interface**, *Proc. of the 3rd Workshop on Object Oriented Programming*, Paris, France, Jan. 1986.

[Hullot 87] **Interface Builder**, a program distributed by ExperTelligence, 559 San Ysidro Road, Santa Barbara, CA. March 1987.

[Manasse 87] M.S.Manasse, G.Nelson: **The Trestle window system**. Digital Equipment Corporation, Systems Research Center, Research Report (to appear).

[McJones 87] P.McJones, G.Swart: **Evolving the Unix system interface to support multi-threaded programs**, Digital Equipment Corporation, Systems Research Center, Research Report (to appear).

[Myers 86] B.A.Myers, W.A.S.Buxton: **Creating highly interactive and graphical user interfaces by demonstration**, *Computer Graphics* Vol.20, no. 4, August 1986, pp 249-258.

[Pfaff 83] G.Pfaff, Ed.: **User interface management systems**, Springer-Verlag, New York, 1985.

[Rovner 85] P.Rovner, R.Levin, J.Wick: **On extending Modula-2 for building large, integrated systems**, Digital Equipment Corporation, Systems Research Center, Technical Report no. 3, January 1985.

[Scheifler 86] R.W.Scheifler, J.Gettys: **The X window system**, *ACM Transactions on Graphics*, Vol. 5, no. 2, April 1986, pp. 79-109.

[Schneiderman 83] B.Schneiderman: **Direct manipulation: a step beyond programming languages**, *IEEE Computer*, Vol 16, no. 8. Aug. 1983. pp. 57-69.

[Schulert 85] A.H.Schulert, G.T.Rogers, J.A.Hamilton: **ADM - a dialog manager**, *Proc. of the conference on Human Factors in Computing Systems*, San Francisco, April 14-18, 1985.

[Smith 83] D.C.Smith, C.Irby, R.Kimball, W.Verplank, E.Harslem: **Designing the Star user interface**, *Byte 7, 4,* April 1982, pp 242-282.

[Thacker 87] C.Thacker, L.Stewart: **Firefly: a multiprocessor workstation**, *Proc of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, CA, October 1987 (to appear).