InterViews: A C++ Graphical Interface Toolkit

Mark A. Linton, Paul R. Calder, and John M. Vlissides
Stanford University

Abstract

We have implemented an object-oriented user interface package, called InterViews, that supports the composition of a graphical user interface from a set of interactive objects. The base class for interactive objects, called an **interactor**, and base class for composite objects, called a **scene**, define a protocol for combining interactive behaviors. Subclasses of scene define common types of composition: a **box** tiles its components, a **tray** allows components to overlap or constrain each other's placement, a **deck** stacks its components so that only one is visible, a **frame** adds a border, and a **viewport** shows part of a component. Predefined components include menus, scrollers, buttons, and text editors. InterViews also includes classes for structured text and graphics. InterViews is written in C++ and runs on top of the X window system.

1 Introduction

Graphical user interfaces are difficult to implement because of diverse user needs and preferences. Tools that assist a graphical interface programmer must balance conflicting requirements carefully. High-level tools can restrict the variety of interfaces that can be created, while tools at too low a level may not provide much help to the programmer.

InterViews (*Inter*active *Views*) is a library of C++[5] classes that can be used to construct a graphical user interface from interactive components. The design of InterViews has been driven by three desires: (1) to avoid constraining the style of the interface, (2) to allow the interface to be defined by composition of existing components, and (3) to allow new components to be derived easily from existing ones.

Like MacApp[4] and Smalltalk MVC[1], the approach used in InterViews separates interactive behavior from abstract behavior. An interactive object, called a **view**, defines the user interface to an abstract object, called the **subject**. The separation of subject and view supports different views of the same subject to suit the particular application or to customize interactive style. A view can be customized dynamically using a **metaview**, a view of another view's internal state. For example, a metaview might allow the user to interactively modify the mapping from keystrokes to commands in a text view.

Building graphical interfaces from reusable components requires the ability to define an interactive object that can be used in a variety of contexts. To fulfill this requirement, we must consider the way in which the characteristics of a component and its context affect each other.

In InterViews, each interactive component, called an **interactor**, has a preferred shape and size. The preferred shape and size of a composition of components, called a **scene**, is calculated from those of the components. However, the actual display space allocated to an interactor might not correspond to its preferred size—the interactor is responsible for making best use of the space it has been allocated. Different scenes allocate display space to component interactors using different algorithms. For example, a **box** tiles its components, but a **tray** allows them to overlap.

We have implemented InterViews on top of the X window system[3]. A small set of primitive classes completely encapsulates the X interface. The remaining library classes and applications do not contain any X calls; they call operations defined by the primitives.

2 Class Organization

Figure 1 shows a subset of the InterViews class hierarchy. Several factors influenced the structure of the hierarchy. The overriding goal was simplicity—to make the classes easy to understand, straightforward to implement, and convenient to extend. From most important to least important, the factors were:

Shallow Nesting
: Classes are a good partitioning mechanism, but there are drawbacks associated with a large number of classes. Our experience has been

[0]Research supported by the SUNDEC project through a gift from Digital Equipment Corporation and a grant from the Charles Lee Powell Foundation.

[0]An early version of this paper titled "The Design and Implementation of InterViews" appears in *Proceedings of the USENIX C++ Workshop*, Santa Fe, New Mexico, November 1987.

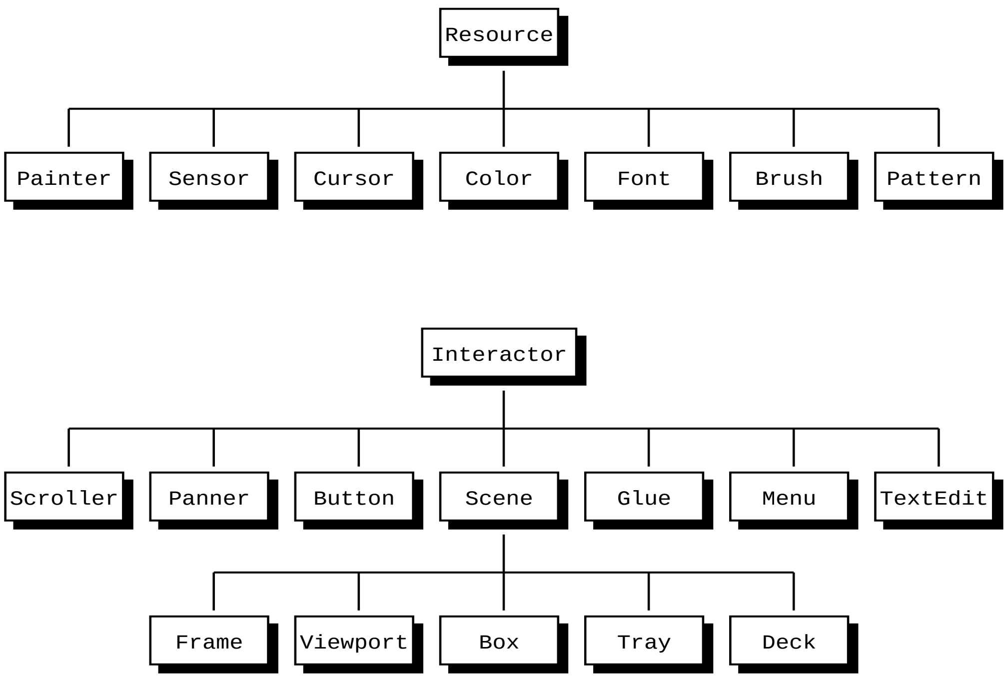

Figure 1: InterViews class hierarchy

that a large class hierarchy overwhelms programmers, especially when there are many levels of subclasses. An earlier library had many small classes nested up to 12 deep. Users of this library had trouble grasping the many different classes and their inherited behavior. InterViews currently has one class nested 5 deep; most classes are at level 2 or 3. Throughout the development of InterViews we often chose to add an operation to an existing class rather than adding a new class.

Object Sharing

One reason for introducing new classes is to allow state to be shared among several objects. For example, several interactors should be able to use the same graphics state; therefore, graphics state is a separate class. Similarly, several graphics states should be able to refer to the same font, so font is a separate class. The top set of classes in Figure 1 are subclasses of **resource**, because they are shared objects. Resources contain a reference count that can be manually incremented. When a resource is destroyed, the reference count is decremented; then the resource is deallocated if the count is zero.

Common Usage

It is often preferable to design for a specific common case than for the general case. For example, InterViews does not define a unifying class for all kinds of menus. Instead, it provides a particular style, pop-up menus. The advantage is that a user need not understand a complicated menu model to use pop-up menus. The disadvantage is that there is no direct support for other kinds of menus, though they are straightforward to implement.

2.1 Interactors

All user interface objects are derived from the interactor class. Every interactor has an associated **shape** that it uses to specify its desired display space allocation. Shapes define a natural size, a shrinkability, and a stretchability. The natural size is the interactor's preferred width and height, the stretchability is the amount by which it is prepared to stretch beyond its natural size, and the shrinkability is the amount by which it is prepared to shrink. An interactor's parent scene may use the shapes of its components to allocate their display area and to determine its own shape. The actual display area allocated to a component interactor is assigned to its **canvas**. If the interactor is a scene, it may in turn allocate parts of its canvas to its own components according to their shapes. Because the current implementation is on top of X, canvases are always rectangular and may overlap.

An interactor defines a set of operations that characterize its behavior. Figure 2 gives the C++ interface to these operations. The `Draw` operation defines the appearance of the interactor. Calling `Draw` causes the interactor to display a representation of itself on its canvas. If the interactor is a scene, it will also call `Draw` on each of its components. `Redraw` is called whenever a part of an interactor needs to be redrawn, perhaps because it had been obscured but is now visible. The `Update` operation indicates that some state on which the interactor depends may have changed; the interactor will usually `Draw` itself in response to an `Update` call. Typically, when a subject changes it will call `Update` on its views. `Reshape` provides a way of controlling an interactor's shape. If an interactor changes its shape, it will usually propagate the change to its parent scene. `Resize` indicates that the interactor's canvas has been changed in size. If the interactor is a scene, it will reallocate its components' canvases.

An interactor performs output to its canvas using a **painter**. A painter provides drawing operations and manages graphics state such as foreground and background colors, font, and fill pattern. Each drawing operation is passed the target canvas. Canvas coordinates refer to pixels but can be expressed in inches or centimeters by multiplying by the predefined global values "inch" or "cm". Also, painters can perform coordinate transformations composed of translations, rotations, and scalings.

A **sensor** defines interest in certain kinds of events. Interactors interested in input events have a sensor that defines their current input interest. Each event is targetted to a particular interactor.

An interactor can receive input events in one of two ways: (1) it can read the next event from the (global) input queue, or (2) an event can be passed from another interactor using the `Handle` operation. The reader of an event may choose to process or ignore the event, or to pass it to the event's target interactor.

A purely event-driven organization, such as in MacApp, can be produced by using the C++ loop

```
for (;;) {
    Read(e);
    e.target->Handle(e);
}
```

A more traditional control flow, not possible in purely event-driven systems, can be produced by reading events as part of interactor operations. For example, when a button is pressed in a pop-up menu it may be desirable to ignore events for targets other than menu items. In cases such as this, reading events directly is more straightforward than an event-driven implementation.

Most interactors handle input and generate output. Thus, every interactor has a sensor called "input" and a painter called "output" for which initial values are defined when the interactor is created. This approach lets interactors define event interest and graphics state dynamically. Interactors can also define additional sensors and painters. For example, an interactor representing a menu selection might use one painter when it is not selected, and another (with reversed colors) to highlight itself when it is selected.

Scenes often pass "input" and "output" to their component interactors, effectively sharing the state among several interactors. Because the state may be shared, it is inconvenient to make a particular interactor responsible for destroying the sensor and painter. The sensor and painter classes are therefore subclasses of resource. The interactor constructor explicitly increments the reference counts of "input" and "output", and the destructor decrements them.

2.2 Scenes

All interactors that contain component interactors are derived from the scene class. Scene subclasses differ primarily in the way their shape depends on the shapes of their components and in the way they allocate display space to their components.

Scenes define operations for managing their components. The C++ interface is given in Figure 3. `Insert` and `Remove` are used to specify a scene's components. An interactor can be a component of only one scene—the interactor structure is a hierarchy. Some scenes have only one component; inserting a component implicitly removes any existing component. `Raise` and `Lower` modify the front-to-back ordering of components within a scene. `Move` suggests a change in the position of a component within the scene. Not all scenes implement all of these operations. For instance, it does not make sense to call `Raise` on a scene that can have only one component.

The `Change` operation tells a scene that one of its components' shapes has changed. A scene can do either of two things in response to a `Change`: it can

```
virtual void Resize();
virtual void Draw();
virtual void Redraw(Coord left, Coord bottom, Coord right, Coord top);
virtual void Reshape(Shape&);
virtual void Update();
virtual void Handle(Event&);
```

Figure 2: Interface to Interactor base class operations

```
void Insert(Interactor*);
void Insert(Interactor*, Coord x, Coord y);
void Change(Interactor*);
void Move(Interactor*, Coord x, Coord y);
void Remove(Interactor*);
void Raise(Interactor*);
void Lower(Interactor*);
void Propagate(boolean);
```

Figure 3: Interface to Scene operations

recalculate its own shape and propagate the change by calling `Change` on its parent, or it can simply reallocate its components' canvases based on the new shape. The `Propagate` operation is used to specify which behavior is required for a particular instance.

2.2.1 Box

Many graphical interfaces can be composed by arranging components side-by-side either horizontally or vertically. The scene subclasses **hbox** and **vbox** support this style of tiled composition. **Glue** provides a way of inserting space between the components in a box. This model is a simplified version of TEX[2] boxes and glue.

A box's shape is the sum of its components' shapes. When allocating its components' canvases, a box tries to allocate each component its natural size. If there is a discrepancy between the available space and the natural size, a box distributes the excess or shortfall according to the proportion of the total stretchability or shrinkability contributed by each component. For example, consider a box which contains one interactor whose shape, expressed as $<natural, shrink, stretch>$, is $<10, 2, 7>$ and another whose shape is $<15, 10, 1>$. If the box is given a canvas of size 25, it can allocate each component its natural size. If the box is given size 19, then it will shrink the first component by $6 \times 2/12 = 1$ and the second component by $6 \times 10/12 = 5$. If the box is given size 33, then it will stretch the first component by $8 \times 7/8 = 7$ and the second component by $8 \times 1/8 = 1$. A shape which is in effect infinitely stretchable or shrinkable can be specified by using the predefined values “hfil” and “vfil.”

Figure 4 illustrates a typical composition using boxes and glue. The upper part of the figure shows two views of an alert box that consists of a vbox containing five components. From the top, they are a piece of glue, an interactor containing the text of the alert, another piece of glue, an hbox containing the button, and a final piece of glue. The hbox is composed of a button with glue on both sides. By suitable choice of the shapes of the various components, the layout of the alert box can be controlled for a range of display sizes. The lower part of the figure shows that, in this instance, the glue to the left of the button is made “infinitely” stretchable and shrinkable, while that to the right is rigid. Thus when the alert is made wider, all of the extra space is absorbed by the glue on the left.

Boxes and glue allow flexible specification of the presentation of a user interface. Many common layout strategies can be expressed easily. For example, a component can be centered within a box by placing “fil” glue on either side, or a number of components can be spaced equally by inserting identically sized glue between them.

2.2.2 Tray

A composition in which components can be placed at specified positions is supported by the scene subclass

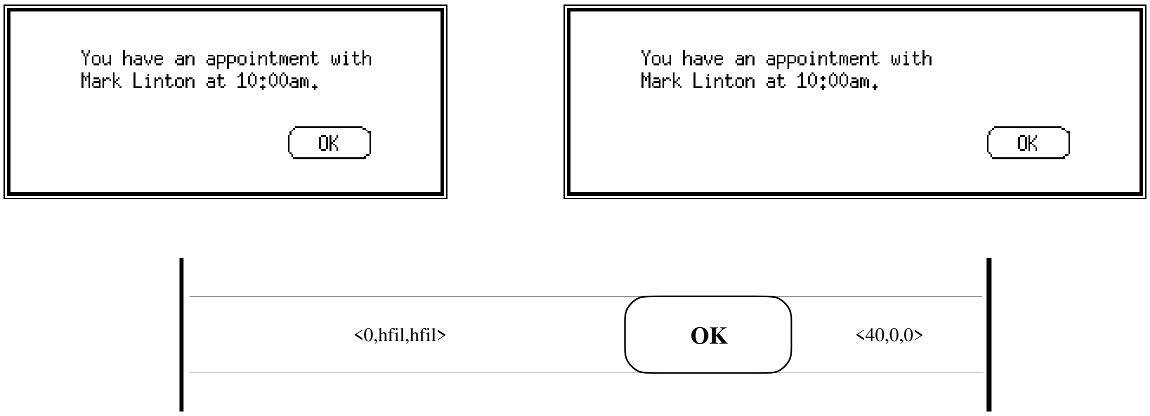

Figure 4: An example of composition with boxes and glue

tray. A tray has a "background" component that is allocated all of the tray's canvas and a number of other components each of whose position is determined by a set of individual alignments. If no explicit alignment is defined for a component, it is assumed to be aligned to the lower left corner of the tray. Components in a tray are arranged from back to front in insertion order and can overlap arbitrarily.

Each alignment of a component interactor is to some other "target" interactor, often another component of the tray or the tray itself. The alignment specifies a point on the target, a point on the component, and the characteristics of the "glue" with which to connect the alignment points. The aligned points can be a corner of the interactor, the midpoint of a side, or the center. The tray will place each component to satisfy its alignments as far as possible. The interactor and the connecting glue will be stretched or shrunk according to their contributions to the total stretchability and shrinkability in the same manner as components within a box.

Trays provide a natural way to describe layouts in which components "float" in front of a background. For example, consider a composition to center a title near the top of a diagram. Figure 5 shows a possible layout. The interactor representing the diagram is the background of the tray, and an interactor containing the title has been inserted with an alignment from the midpoint of its top edge to the midpoint of the tray's top edge. The natural size of the glue connecting the aligned points determines the distance of the title from the top edge of the diagram. Other examples of layouts easily described using a tray include "pull down" menus where the menu is aligned to a fixed "menu bar" and transient "alert boxes" which are often centered atop another interactor.

2.2.3 Deck

A third style of composition is provided by the scene subclass **deck**. Components in a deck are conceptually stacked on top of each other so that only the topmost component is visible at a time. A deck takes the shape of the largest of its components and allocates all of its canvas to the topmost component. A set of operations provide the means to "shuffle" the deck to bring the desired component to the top. The visible component can also be selected interactively using a scroll bar.

A deck is useful in composing a layout such as a multi-page document in which each page is represented by an interactor in the deck. Another use might be to compose a "dialog" in which there are several alternate "panels" of options—a deck could be used to switch between the panels.

2.2.4 Single Component Scenes

Graphical interfaces commonly require interactors that are best described using another interactor. For example, a menu is implemented as a box containing menu items. However, a menu does not share the behavior of a box in the sense of a subclass; it simply uses the box to compose the items. This distinction is important, and it helps simplify the class hierarchy. In InterViews such interactors are implemented using the scene subclass **monoscene**, which can contain only a single component. A monoscene normally gives all

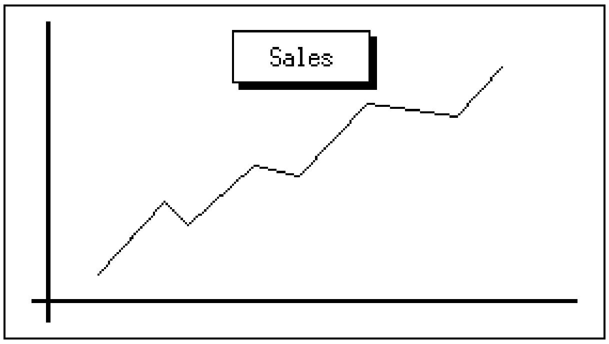

Figure 5: A layout using a tray

of its display space to the interactor. One subclass of monoscene, **frame**, allocates all of its display space except for an outline around the interactor. For example, the alert box of Figure 4 has a frame around the vbox which composes the inner components. Another subclass, **viewport**, allows its component to be larger than the available space. Part of the component is visible through the viewport; how much is visible can be controlled by the user through interactors such as scroll bars.

2.3 Perspectives

InterViews provides a standard way to handle scrolling, zooming, and panning operations on an interactor. An interactor that implements such operations maintains a **perspective**. The perspective defines a range of coordinates representing the total extent of the interactor's view and a subrange for the portion of the total range that is currently visible. For example, the vertical range for a text editor might be the total number of lines in a file; the subrange would be the number of lines actually displayed in the editor's canvas.

Scrolling and zooming are performed by modifying the interactor's perspective. An interactor can modify its own perspective (when the text editor adds a line to the file, for example), or the perspective can be modified as a result of an external operation such as a user request.

A **scroller** is an interactor that is a view of the perspective associated with another interactor. A scroller displays a sliding bar whose length reflects the fraction of the total range that is currently visible. The user can modify the perspective interactively through the scroller using the mouse.

Other kinds of views of perspectives are also provided by InterViews: a **panner** supports movement in both x and y dimensions from a single interface, and zoom and scroll buttons allow stepped adjustments. Several views of the same perspective can exist at once. For instance, a perspective could be modified by zoom and scroll buttons in addition to a scroller. When a perspective is changed, it notifies its views. Thus, a change made through one view of a perspective will be reflected in all of its views.

2.4 Buttons

A **button** is an interactor subclass that is a view of a **button state**. The user can "press" a button to set the associated button state to a particular value. Several buttons can be visible for the same button state, making it possible to use buttons to select from a discrete set of values, each button representing a different value. Like any subject, a button state notifies its views (buttons) when it changes.

Three common kinds of buttons are provided. A **push button** has a round-cornered rectangle surrounding its label. It is drawn in reverse colors when it is pressed. A push button remains pressed only as long as the user hold down the mouse button. A **radio button** has a circle to the left of its label; the circle is filled when the button is pressed. A radio button acts like a tuning button on a car radio. Pressing the button sets the associated button state to a particular value. The button will stay pressed until the button state is changed to a new value, usually by pressing another radio button for the same button state. A **check box** has a square to the left of its label; the square is checked when the button is pressed. A check box has a "push-on/push-off" action. Successive clicks of the mouse button alternately press and release the check box button. Figure 8 shows all three types of button composed in a dialog box.

In addition to being attached to a button state, buttons can be attached to other buttons. If button A is attached to button B, then A is disabled while B is not pressed. A disabled button ignores input and draws itself "grayed out" to show that it is disabled.

The button model implemented by the InterViews library is well suited to a dialog mode of user interaction—buttons are used to set state variables but cause no immediate action. Only when the dialog is dismissed, often with a push button, are the current values of the state variables examined and interpreted. Other button models, such as a button with an associated action rather than state, can be derived easily.

2.5 Structured Text

InterViews provides **text** objects that can be structured hierarchically using composition objects that support a variety of layout styles. The **composition** class defines the way objects of class text are arranged to fill available space. Subclasses of composition specify different layout strategies. There is a close parallel between a composition containing text objects and a scene containing interactors. In both cases, subclasses of a general composition class determine the way components are laid out to fill available space.

Primitive textual objects derived from the text class include **word** and **whitespace**. The **edit word** subclass provides character editing operations such as `Insert` and `Delete`. Subclasses of composition include **sentence**, which causes a line break at the right margin; **paragraph**, which defines right and left margins; **text list**, which composes components horizontally or vertically depending on the available space; and **display**, which indents its components.

Conceptually, text objects form one long line of text. Composition objects can be used to place constraints on how this line is broken to fit available space. For example, a text list object will arrange text objects horizontally if there is enough room; otherwise it will place each of its components on a separate line.

Composition objects also support hit detection by returning the text object corresponding to a coordinate pair. This facility lets views determine which text object is selected without knowing the layout.

2.6 Structured Graphics

Graphic is a base class for defining structured graphics objects. Each graphic has its own graphics state that includes attributes such as color, line style, and coordinate transformation. Subclasses of graphic include **line**, **circle**, **rectangle**, and **picture** (for representing a collection of graphics). All graphics can draw and erase themselves and provide operations for examining and changing graphics state attributes.

Pictures are the basic mechanism for building hierarchies of graphics. A picture maintains a list of component graphics and draws itself by drawing each component with a graphics state formed by combining the component's state with its own. For example, combining color attributes means the component's color is overridden by the picture's (if the picture defines one); combining coordinate transformations means multiplying the component's transformation matrix by the picture's. This scheme makes operations on a picture affect its components so that an operation works on the picture as a unit.

Graphics also support hit detection; for instance, a hit can be registered on a **spline** object within one pixel. Pictures perform hit detection by checking for hits on their component graphics in the picture's coordinate space.

Damage is a class that automatically redraws portions of a graphic that have been changed, erased, or are otherwise inconsistent with the graphic's intended appearance. Damage objects try to minimize the amount of redraw needed to repair a graphic. They are most useful for repairing graphics that are complicated enough to make redrawing the entire canvas undesirable.

2.7 Other Classes

Rubberband is a base class for graphics objects that track user input. For example, a **rubber rectangle** can be used to drag out a new rectangle interactively. Another subclass, **sliding rectangle**, can be used to move around an existing rectangle. These classes completely isolate programmers from device-dependent methods commonly used to implement "rubberbanding", such as use of exclusive-or drawing or an overlay plane.

A **bitmap** represents a bit-mapped mask suitable for drawing icons or for constructing other objects such as cursors and fill patterns. Operations such as scaling, rotation, and bit manipulations are provided.

Every program using InterViews must create a **world** object. This object represents the root scene of the display. The constructor opens a connection to the window server. Other InterViews classes include **banner**, which displays headings, **border**, which visually separates components in boxes, and **menu**, which is a box of **menu items** that inserts itself into the world when its `Popup` operation is called. After insertion, a menu waits for the user to release a button and then returns the menu item that was chosen.

3 Example Usage

Squares is a demonstration program that uses many of the InterViews classes. The program contains a simple subject that manages a list of squares of different sizes and positions. The user interface is constructed from a view of the squares list, a frame around the view, and a metaview for simple customization.

The frame surrounds a vertical box containing a banner and two horizontal boxes, all separated by horizontal borders. The upper horizontal box contains the squares view, a vertical border, and a vertical scroller. The lower horizontal box contains a horizontal scroller, a vertical border, and a piece of glue. Figure 6 shows what the squares frame looks like, and Figure 7 shows the C++ code that constructs the frame.

Using a pop-up menu, the user can create another view of the squares list, add a square to the list, open a metaview to customize the squares frame, or exit the program. The squares list notifies its views when the square is added, so the new square is visible in all views. Each view can be scrolled and zoomed independently.

Figure 8 shows the metaview used to customize the frame around a view. The metaview consists of a dialog box containing check boxes for specifying the presence of scrollers, buttons for specifying attributes of the scrollers, and a confirmation button to indicate that customization is complete. The components of the dialog box are separated by glue objects with carefully chosen shapes; the dialog will maintain a pleasing layout for a range of sizes.

4 Implementation

It took about six man-months to implement the initial version of InterViews on top of X. In this section, we discuss some of the problems in interfacing to X, some details of implementing scenes, and some comments on using C++ and object-oriented programming in general.

4.1 Interfacing to X

InterViews primitive class operations make direct X library calls to implement their semantics. The key issues in interfacing to X were managing X windows and translating X input events into InterViews events.

4.1.1 Window Management

Each canvas is represented as an X window. The world's canvas is the root window for a display. The scene class contains operations to handle the creation, mapping, and configuration of windows. The two operations available for use by scene subclasses when allocating component interactors' canvases are `Place` and `UserPlace`. `Place` puts an interactor at a specific position in a scene and is implemented by creating a subwindow of the scene's window and associating the subwindow with the interactor's canvas. `UserPlace` creates a window and lets the user interactively position it.

4.1.2 Input Events

The X model of input events is somewhat different from the InterViews model, but an important similarity is that each X input event is associated with a destination window. The interactor `Read` operation maps the window to the target through a global hash table maintained by scenes. The event is then checked against the interactor's current sensor to see if the interactor is interested in the event. Normally, we can tell X to ignore events that are not of interest; however, X cannot always distinguish events at the level we wish. For example, X cannot send events for the left mouse button and ignore events for the middle and right buttons.

X is very different from InterViews in the sizing and redrawing of windows. X represents the need to redraw part of a window as an input event; InterViews represents it as an out-of-band procedure call. When the `Read` operation sees a redraw event, it calls `Redraw` on the destination window and proceeds to read the next input event.

4.2 Scene Shapes

An important aspect of implementing scenes is the computation of the scene's shape. A graphical interface designer should be free to concentrate on the components and rely on the scene to compose their shapes appropriately.

For example, consider the calculation of the shape of a box. A box must compute its own shape as a function of the shapes of the interactors inside it. Along the major axis (horizontal for an hbox, vertical for a vbox), the natural sizes, stretchabilites and shrinkabilities can simply be added.

Computing the parameters for the minor axis is more complicated. The model we adopted is that the box

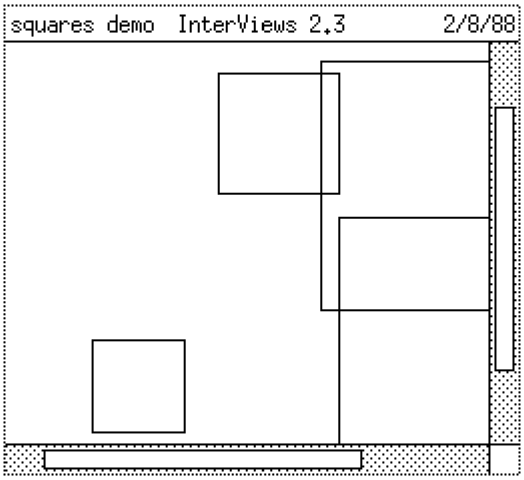

Figure 6: Squares view

```
frame = new VBox(
    new Banner("squares demo", "InterViews 2.3", "2/8/88"),
    new HBorder,
    new HBox(view, new VBorder, new VScroller(view, vwidth)),
    new HBorder,
    new HBox(new HScroller(view, hwidth), new VBorder, new HGlue(vwidth, 0))
);
```

Figure 7: Code to construct squares frame

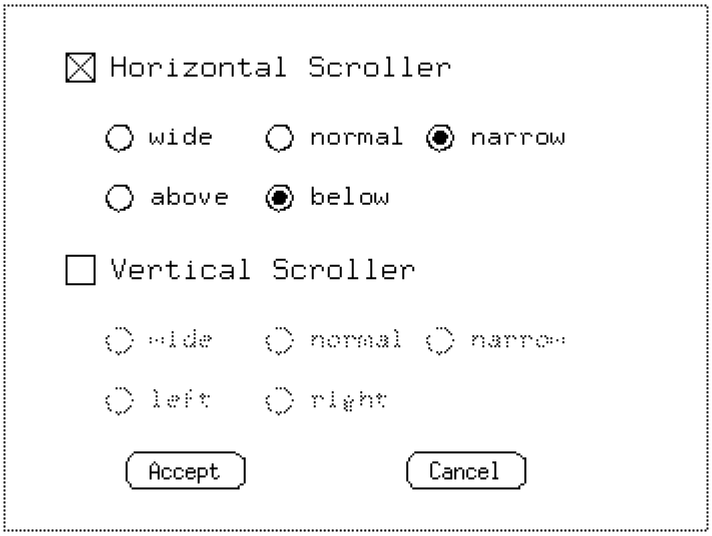

Figure 8: Squares metaview dialog

should be big enough to accommodate the largest component and should stretch and shrink no more than the most rigid component will allow. If these requirements conflict, the larger number is used. Letting N, L, H, α, β represent natural size, minimum size, maximum size, shrinkability, and stretchability, respectively, then for a set of k components the parameters are calculated as follows:

$$N_{minor} = max(N_1, \ldots, N_k)$$

$$L_{minor} = max(L_1, \ldots, L_k)$$

where

$$L_i = N_i - \alpha_i,$$

and

$$H_{minor} = min(H_1, \ldots, H_k)$$

where

$$H_i = N_i + \beta_i.$$

4.3 Experience with C++

Using C++ as the implementation language for InterViews has had several benefits. Class inheritance and virtual functions simplify the structure of code and data, making the implementation easier to debug and understand. Much of the complexity is in the primitive classes, hidden from interface designers. Ease of understanding is especially important for InterViews, since it is intended that interface designers will derive application-specific classes from library classes to suit particular needs. C++ is also portable, enabling us to bring up InterViews on a new workstation quickly.

A significant advantage of using C++ for InterViews was that there was a good match between the language and the software we were designing. It was much easier to implement an object-oriented user interface package using an object-oriented language than it would have been with a procedural language. Classes define objects that model closely the real objects and concepts the system is meant to manage. The programmer focuses on the objects that are manipulated, not on the flow of control. In fact, an earlier version of InterViews was implemented in Modula-2. Rewriting the code in C++ resulted in a considerably cleaner implementation.

An observation we made during the design of InterViews was that it is important to concentrate on the *protocols* for communication between objects. If these protocols are well designed, then the implementation of the objects is relatively straightforward. Conversely, if consideration is not given to such issues, much reorganization of class hierarchies and considerable recoding is likely to result.

5 Current Status

InterViews currently runs on MicroVAX and Sun workstations on top of either X10 or X11. The library is roughly 25,000 lines of C++ source code. We have also implemented several applications on top of the library, including a reminder service, a scalable digital clock, a drawing editor, a load monitor, a window manager, and a display of incoming mail. The applications have been used daily by about 20 researchers for a year, and the library is being used in many development efforts at Stanford and other universities, and in industry. We are currently working on a more general drawing system, a program structure editor, and a visual debugger.

6 Conclusion

InterViews provides a simple organization of graphical interface classes that is easy to use and extend via subclassing. Scene subclasses such as box, tray, deck and frame make it possible to compose interactive components into complete interfaces without specifying layout details. Abstract and interactive behavior are separated into subject and view objects to support different interfaces to the same functionality.

The InterViews library completely hides the underlying window system from application programs. This means that we can port InterViews applications to a new window system simply by porting the primitive classes.

Using an object-oriented language to implement InterViews resulted in a package that is both simple to use and easy to extend. The interface designer is encouraged to think in object-oriented terms, usually the most natural way of expressing interactive behavior.

References

[1] Goldberg, A., *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, Reading, Massachusetts, 1984.

[2] Knuth, D., *The TEXbook*, Addison-Wesley, Reading, Massachusetts, 1984.

[3] Scheifler, R.W., and J. Gettys, "The X Window System", *ACM Transactions on Graphics* Vol. 5, No. 2, April 1986, pp. 79-109.

[4] Schmucker, K. J., *Object-Oriented Programming for the Macintosh*, Hayden, Hasbrouck Heights, New Jersey, 1986.

[5] Stroustrup, B., *The C++ Programming Language*, Addison-Wesley, Reading, Massachusetts, 1986.