

Composing User Interfaces with InterViews

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

Graphical user interfaces for workstation applications are inherently difficult to build without abstractions that simplify the implementation process. To help programmers create such interfaces, we considered the following questions: What sort of interfaces should be supported? What constitutes a good set of programming abstractions for building such interfaces? How does a programmer build an interface given these abstractions? Practical experience has guided our efforts to develop user interface tools that address these questions. We make the following observations:

All user interfaces need not look alike. It is desirable to maintain a consistent "look and feel" across applications, but users often have different preferences. For example, one user may prefer pop-up menus, while another insists on pull-down menus. Our tools must therefore allow a broad range of interface styles and must be customizable on a per-user basis.

User interfaces need not be purely graphical. Many application designers prefer iconic interfaces because they believe



The InterViews toolkit offers a rich set of composition mechanisms and a variety of predefined objects, allowing easy implementation of complex user interfaces.

novices understand pictures more readily than text. However, recent work¹ suggests that excessive use of icons can confuse the user with unfamiliar symbolism. A textual interface might be more appropriate in a given context. The choice of graphical or

textual representation should favor the clearest alternative.

User interface code should be object-oriented. Objects are natural for representing the elements of a user interface and supporting their direct manipulation. Objects provide a good abstraction mechanism, encapsulating state and operations, and inheritance makes extension easy. Our experience shows that, compared with a procedural implementation, user interfaces written in an object-oriented language are significantly easier to develop and maintain.

Interactive and abstract objects should be separate. Separating user interface and application code makes it possible to change the interface without modifying the underlying functionality, and vice versa. This separation also facilitates customization by allowing several interfaces to the same application. It is important to distinguish between interactive objects, which implement the interface, and abstract objects, which implement operations on the data underlying the interface.

An effective way to support these principles is to equip programmers with a

toolkit of primitive user interface objects that use a common protocol to define their behavior. The protocol allows uniform treatment of user interface objects, enabling in turn the introduction of objects that compose primitives into complete interfaces. Different classes of composition objects can provide different sorts of composition. For example, one class of composition object might arrange its components in abutting or *tiled* layouts, while another allows them to overlap in prescribed ways. A rich set of primitive and composition objects promotes flexibility, while composition itself represents a powerful way to specify sophisticated and diverse interfaces.

Composition mechanisms are central to the design of InterViews, a graphical user interface toolkit we have developed at Stanford. InterViews is a library of C++ classes that define common interactive objects and composition strategies. Figure 1 depicts how objects from the InterViews library are incorporated into an application, and Figure 2 shows the relationship between the various layers of software that support the application. Primitive and composition objects from the library are linked into application code. The window system is entirely abstracted from the application; the application's user interface is defined in terms of InterViews objects, which communicate with the window and operating systems.

InterViews supports composition of three object categories. Each category is implemented as a hierarchy of object classes derived from a common base class. Composition subclasses within each class hierarchy allow hierarchical composition of object instances.

(1) Interactive objects such as buttons and menus are derived from the *interactor* base class. Interactors are composed by scenes; scene subclasses define specific composition semantics such as tiling or overlapping.

(2) Structured graphics objects such as circles and polygons are derived from the *graphic* base class. Graphic objects are composed by pictures, which provide a common coordinate system and graphical context for their components.

(3) Structured text objects such as words and whitespace are derived from the *text* base class. Text objects are composed by clauses; clause subclasses define common strategies for arranging components to fill available space.

The base classes define the communication protocol for all objects in the hierar-

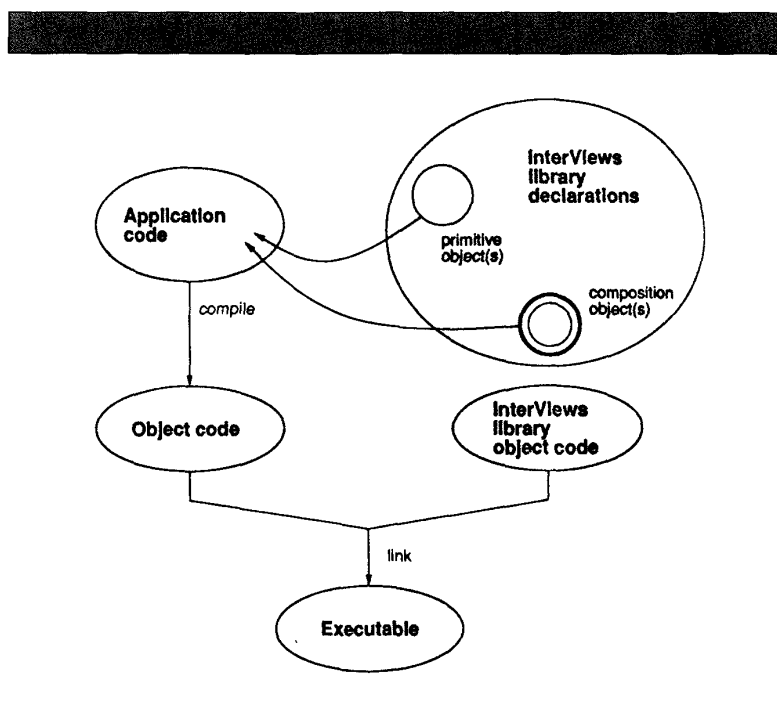


Figure 1. Incorporating InterViews objects into an application.

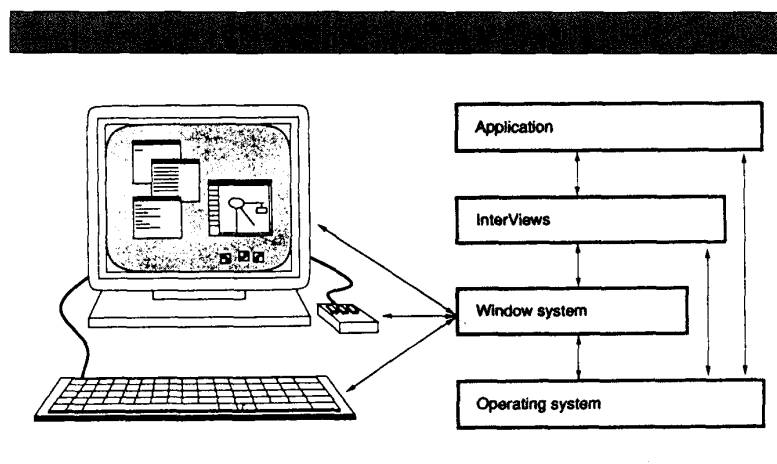


Figure 2. Layers of software underlying an application.

chy. The composition classes define the additional protocol needed by the elements in a composition, such as operations for inserting and removing elements and for propagating information through the composition (see the sidebar entitled "Primitive and composition protocols").

Hierarchical composition gives the programmer considerable flexibility. Com-

plex behavior can be specified by building compositions that combine simple behavior. The composition protocol facilitates the tasks of both the designer of a user interface toolkit and the implementor of a particular user interface. The toolkit designer can concentrate on implementing the behavior of a specific component in isolation; the interface designer is free to

combine components in any way that suits the application.

In this article we focus on using InterViews to build user interfaces. We present several simple applications and show how InterViews objects can implement their interfaces. We also illustrate the benefits of separating interactive behavior and abstract data in several different contexts. Finally, we discuss InterViews support for end-user customization as well as the status of the current implementation.

Interactor composition

An interactor manages some area of potential input and output on a workstation display. A scene composes a collection of one or more interactors. Because a scene is itself an interactor, it must distribute its input and output area among its components. In this section, we discuss the various InterViews scene subclasses that provide tiling, overlapping, stacking, and encapsulation of components. We concen-

trate on how these scenes are used rather than giving their precise definitions.

Boxes and glue. Consider the simple dialog box shown in Figure 3. It consists of a string of text, a button containing text, and a white rectangular background surrounded by a black outline. Pushing the button will cause the dialog box to disappear. The dialog box will maintain a reasonable appearance when a window manager resizes it. If parts of the dialog

Primitive and composition protocols

We can think of the set of operations defined on an object as a communication protocol that the object understands. Since objects cannot access the internal state of other objects, interobject dependencies are limited by the semantics of the protocol. Objects are thus isolated from one another, promoting modularity and reusability. Furthermore, objects derived from a common base class (thus obeying a common protocol) can be used without knowledge of their specific class; operations redefined by the subclass are automatically invoked on the objects instead of the corresponding base class operations (a form of dynamic binding). A common protocol allows composition objects to treat their components uniformly. Dynamic binding lets composition objects take advantage of subclass-specific behavior without modification. Together, these attributes make composition possible.

Interactor protocol. The protocol for interactors includes the following operations:

- void Draw();
- void Redraw(Coord left, Coord bottom, Coord right, Coord top);
- void Resize();
- void Update();
- void Handle(Event&);
- void Read(Event&);

The Draw operation defines the interactor's appearance. A call to Draw causes the interactor to draw itself in its entirety. Redraw is called whenever a part of an interactor needs to be redrawn (for example, when it had been obscured but is now visible). A call to Resize notifies the interactor that the screen space it occupies has changed size. The interactor can then take whatever action is appropriate. Draw, Redraw, and Resize are automatically called by InterViews library code in response to window system requests. 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.

Interactors handle input events with the Handle operation. Each event is targeted to a particular interactor. Any interactor can Read the next event from the global event queue. Handle and Read can be used to create event-driven input handling, in which only one interactor is responsible for reading events and forwarding them to their target.

Scene protocol. Scenes add several operations for component management to the basic interactor protocol:

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

Insert and Remove specify a scene's components. Raise modifies the front-to-back ordering of components within a scene to bring the specified component to the top. Move suggests a change in the position of a component within the scene. Not all scenes implement all these operations. For instance, it does not make sense to call Raise on a monoscene, since it can have only one component.

The Change operation tells a scene that one of its components has changed. A scene can do one of two things in response to a Change: It can propagate the change by calling Change on its parent, or it can simply reallocate its components' screen space. The Propagate operation specifies which behavior is required for a particular instance.

Graphic protocol. The graphic base class defines the protocol for drawing objects, manipulating graphics state, and hit detection. Operations include:

- void Draw(Canvas*);
- void DrawClipped(Canvas*, Coord, Coord, Coord, Coord);
- void Erase(Canvas*);
- void EraseClipped(Canvas*, Coord, Coord, Coord, Coord);

- void SetColors(PColor* f, PColor* b);
- void SetPattern(PPattern*);
- void SetBrush(PBrush*);
- void SetFont(PFont*);

- void Translate(float dx, float dy);
- void Scale(float sx, float sy, float ctrx = 0.0, float ctry = 0.0);
- void Rotate(float angle, float ctrx = 0.0, float ctry = 0.0);
- void SetTransformer(Transformer*);

- void GetBounds(float&, float&, float&, float&);
- boolean Contains(PointObj&);
- boolean Intersects(BoxObj&);

box previously covered by other windows are exposed, then the newly exposed regions will be redrawn.

InterViews provides abstractions that closely model the elements, semantics, and behavior of the dialog box. A user interface programmer can express the implementation of the interface in the same terms as its specification. The InterViews library contains a variety of predefined interface components. In the dialog box, we will use *message*, *push button*, *box*,

glue, and *frame*. (See the sidebar entitled "Glossary" for definitions of these terms.)

We use boxes and glue to compose the other elements of the dialog box. The composition model is a simplified version of the TeX³ boxes and glue model. This model makes it unnecessary to specify the exact placement of elements in the interface, and it eliminates the need to implement resize behavior explicitly.

Two types of box are used: An hbox tiles its components horizontally, while a vbox



Figure 3. A simple dialog box.

In addition to the operations for setting graphics state attributes and coordinate transformations, there are complementary operations for obtaining the current values of these parameters. The *Contains* and *Intersects* operations determine whether a user clicked on a graphic. *PointObj* and *BoxObj* specify a point and a rectangular region, respectively. *Contains* can detect an exact hit on a graphic; *Intersects* can detect a hit within a certain tolerance.

Picture protocol. Each picture maintains a list of component graphics. A picture draws itself by drawing each component with a graphics state formed by concatenating the component's state with its own. Pictures define default semantics for concatenation; subclasses of picture can redefine the semantics or rely on their components to do the concatenation.

Contains, *Intersects*, and bounding box operations defined in the graphic base class are redefined in the picture class to consider all the components relative to the picture's coordinate system. The picture class defines operations for editing and traversing its list of components. Pictures also define operations for selecting graphics they compose based on position:

- `Graphic * FirstGraphicContaining(PointObj&);`
- `Graphic * FirstGraphicIntersecting(BoxObj&);`
- `Graphic * FirstGraphicWithin(BoxObj&);`

- `Graphic * LastGraphicContaining(PointObj&);`
- `Graphic * LastGraphicIntersecting(BoxObj&);`
- `Graphic * LastGraphicWithin(BoxObj&);`

- `int GraphicsContaining(PointObj&, Graphic **&);`
- `int GraphicsIntersecting(BoxObj&, Graphic **&);`
- `int GraphicsWithin(BoxObj&, Graphic **&);`

The `...` *Containing* operations return the graphics containing a point; `...` *Intersecting* operations return the graphics intersecting a rectangle; `...` *Within* operations return the graphics falling completely within a rectangle.

Pictures draw their components starting from the first component in the list. The `Last...` operations can select the "topmost" graphic in the picture, while `First...` operations select the "bottommost."

Text protocol. The text object protocol includes the following operations:

- `void Draw(Layout *);`
- `void Locate(Coord &x1, Coord &y1, Coord &x2, Coord &y2);`
- `void Reshape();`

Draw defines the appearance of an object in a given layout. A *Layout* object defines the area of the screen into which a hierarchy of text objects will be composed. *Locate* is used for hit detection on text objects. *Reshape* calculates geometric information about an object for use in implementing composition strategies.

Clause protocol. Clauses add operations for stepping through components and for modifying the list of components:

- `Text * First();`
- `Text * Succ(Text *);`
- `Text * Pred(Text *);`
- `boolean Follows(Text *, Text *);`

- `void Append(Text *);`
- `void Prepend(Text *);`
- `void InsertAfter(Text * old, Text *);`
- `void InsertBefore(Text * old, Text *);`
- `void Replace(Text * old, Text *);`
- `void Remove(Text *);`

First returns the leftmost or topmost component. *Succ* and *Pred* return the successor or predecessor of a component. *Follows* can determine if one component comes before or after another.

To probe further. We have only considered the basic elements of the various protocols in this discussion. A more detailed look at these protocols and the implementations behind them can be found elsewhere.^{1,2}

References

1. M.A. Linton, P.R. Calder, and J.M. Vlissides, *InterViews: A C++ Graphical Interface Toolkit*, Tech. Report CSL-TR-88-358, Stanford Univ., Stanford, Calif., July 1988.
2. J.M. Vlissides and M.A. Linton, "Applying Object-Oriented Design to Structured Graphics," *Proc. 1988 USENIX C++ Conf.*, Oct. 1988, pp. 81-94; also available as Tech. Report CSL-TR-88-364, Stanford Univ., Stanford, Calif., Aug. 1988.

Glossary

box, hbox, vbox Scenes that support tiled composition of interactors.

button The button base class defines the interface to generic button interfaces. *Push buttons* provide a momentary contact interface. *Radio buttons* allow the user to select one of several mutually exclusive choices.

button state A subject that maintains state associated with one or more buttons.

clause The base class for structured text composition objects.

deck A scene that stacks interactors.

display A clause that defines an indented text layout.

frames Monoscenes that embellish their component. Frames add a simple border, *shadow frames* add a drop shadow, and *title frames* add a banner.

glue, hglue, vglue Interactors that act as spacers between components of a scene.

graphic Base class for structured graphics objects.

graphic block An interactor that displays a structured graphics object.

immediate-mode graphics A graphics model in which individual geometric shapes are drawn by routines that simply modify pixels on the screen as they are called.

interactor The base class for interactive objects such as menus and buttons.

message An interactor that displays a string of characters.

mover An interactor that scrolls another interactor by some increment.

painter An object providing immediate-mode graphics operations and operations for setting graphics state parameters.

panner An interactor that supports continuous two-dimensional scrolling and incremental scrolling and zooming.

perspective A subject that maintains scrolling and zooming information, including the total size of a view and how much is currently visible.

phrase A clause that places its components end-to-end on a single line.

picture The base class for structured graphics composition objects.

rectangle A graphic that represents and draws a rectangle.

scene The base class for objects that compose interactors; *monoscenes* are scenes that contain only one component.

sentence A clause that places as many of its components as possible on the same line and begins a new line if necessary.

slider A two-dimensional scroll bar.

structured graphics A graphics model that supports hierarchical composition of graphical elements; support is usually provided for coordinate transformations, hit detection, and automatic screen update.

structured text A graphics model that allows hierarchical composition of textual elements, emphasizing the arrangement of elements to make use of available space.

subject An object that maintains state and operations that underlie a user interface; a subject maintains a list of views to be notified when the subject's state changes.

text The base class for structured text objects.

text block An interactor that displays a structured text object.

text list A clause that arranges its components either horizontally or vertically depending on available space.

tray A scene that maintains constraints on the placement of potentially overlapping components.

view An object that provides the user interface to a subject.

viewport A monoscene that can scroll and zoom its component.

whitespace A text object used to introduce space between other text objects in a clause.

word A text object that represents and draws a string of characters.

zoomer An interactor that magnifies or reduces another interactor.

tiles them vertically. Glue between interactors in a box provides space between components. We use *hglue* in *hboxes* and *vglue* in *vboxes*.

Each interactor defines a preferred or *natural size* and the amount it can stretch

or shrink to fill available space. We can use glue of various natural sizes, shrinkabilities, and stretchabilities to describe a wide variety of interface layouts and resize behaviors.

Figure 4 depicts schematically how the

elements of the dialog box are composed using boxes and glue. The corresponding object structure is shown in Figure 5, and the C++ code that implements the dialog box appears in Figure 6. The message and button interactors are each placed in an

hbox with hglue on either side of them. The hglue to the left of the message has a natural size of $\frac{1}{4}$ -inch and cannot stretch, while the glue on the right has a natural size of zero and can stretch infinitely (as specified by the constant *hfil*). If the dialog box is resized as in Figure 7, the margin to the left of the message will not exceed $\frac{1}{4}$ -inch, while the space to the right can grow arbitrarily. Similarly, the button has infinitely stretchable hglue to its left and fixed-size hglue to its right, so that the margin to the right of the button will not exceed $\frac{1}{4}$ -inch.

The hboxes are composed vertically within a vbox, separated by pieces of vglue. The pieces of vglue above the message and below the button have a natural size of $\frac{1}{4}$ -inch, while the vglue between the message and the button has a natural size of $\frac{1}{2}$ -inch. The inner vglue can stretch twice as much as the outer two pieces of vglue. When resized, therefore, the message and button interactors will remain twice as far apart from each other as they are from the edge of the dialog box.

Tray. Suppose we want a dialog box centered atop another interactor, perhaps to notify the user of an error condition. Furthermore, we want the dialog box to remain centered if the interactor is resized or repositioned. Boxes and glue are inappropriate for this type of nontiled composition.

The *tray* scene subclass provides a natural way to describe layouts in which components “float” in front of a background. A tray typically contains a background interactor and several other components whose positions are determined by a set of alignments. For example, the background interactor might display the text in a document; other components could include various messages, buttons, and menus.

Each alignment of a tray component is to some other target interactor, which can be 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 that connects the alignment points. An alignment point can be a corner of the interactor, the midpoint of a side, or the center. The tray will arrange the components to satisfy all alignments as far as possible. If necessary, the components and the connecting glue will stretch or shrink to satisfy the alignments.

Figure 8 shows a simple application in which a tray composes a textual interface and a dialog box. The interactor contain-

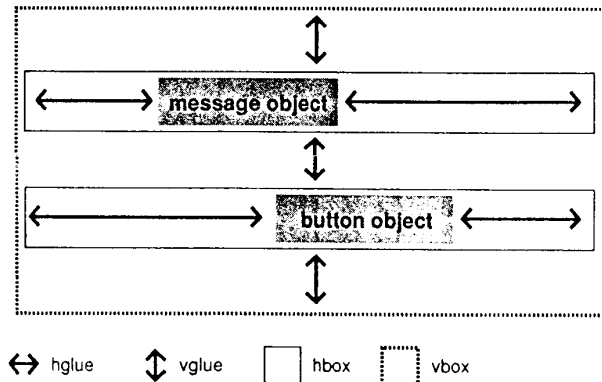


Figure 4. Schematic of dialog box composition using boxes and glue.

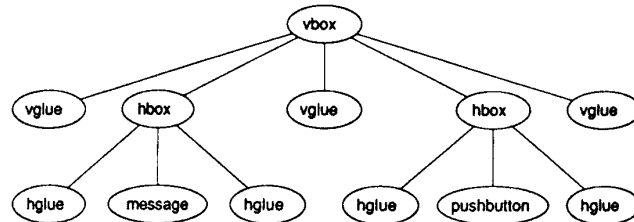


Figure 5. Object structure of dialog box composition.

```
const int space = round(.25*inches);
ButtonState * status;

Frame * frame = new Frame(
    new VBox(
        new VGlue(space, vfil), /* (natural size, stretchability) */
        new HBox(
            new HGlue(space, 0),
            new Message("hello world"),
            new HGlue(0, hfil)
        ),
        new VGlue(2*space, 2*vfil),
        new HBox(
            new HGlue(0, hfil),
            new PushButton("goodbye world", status, false),
            new HGlue(space, 0)
        ),
        new VGlue(space, vfil)
    )
);
```

Figure 6. C++ code for composing the dialog box interface.

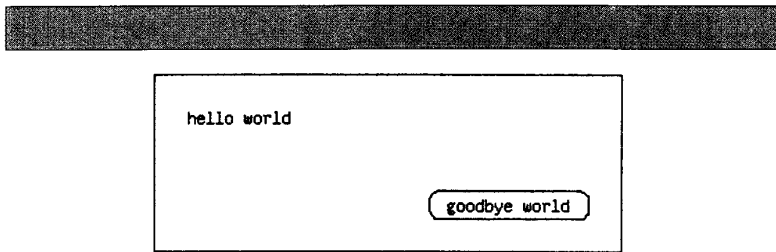


Figure 7. The dialog box after resizing.

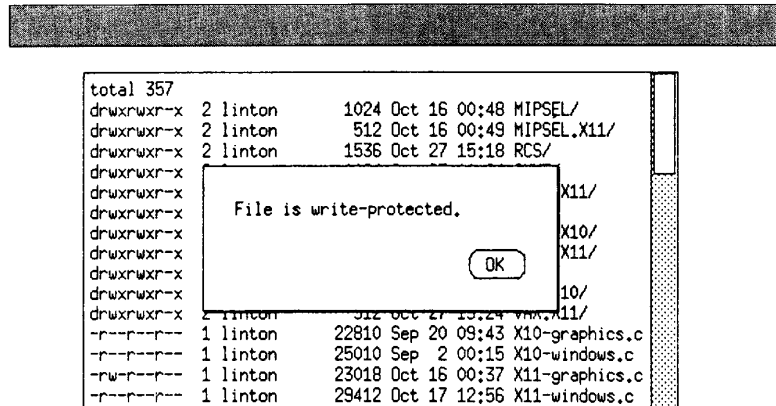


Figure 8. An interface using a tray.

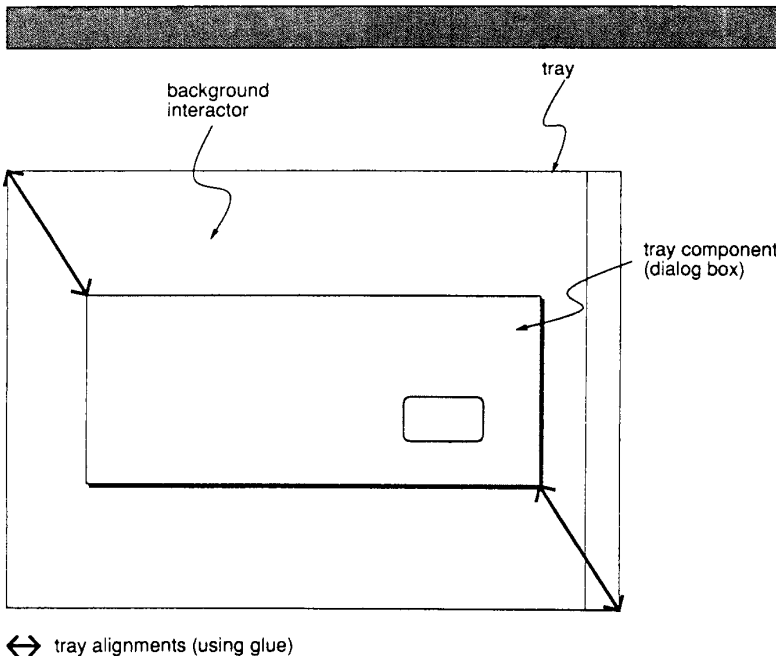


Figure 9. Schematic of tray interface.

ing text and a scroll bar are composed with an hbox and placed into the tray as its background. When the dialog box is required, it is inserted into the tray with its upper left and lower right corners aligned to the corresponding corners of the tray. Figure 9 shows the arrangement of components, and Figure 10 gives the code that implements the interface. The alignments interpose stretchable but nonshrinkable glue with a natural size of $\frac{1}{4}$ -inch to maintain a minimum spacing between the edges of the tray and the dialog box. These alignments guarantee that the dialog box will remain centered atop the background interactor after resizing (see Figure 11). Note how the tray shrank the dialog box to satisfy the alignment constraints once the glue reached its minimum size.

Deck. Another common interface allows the user to flip (rather than scroll) through “pages” of text or graphics as through a book. We can build such an interface in InterViews by composing interactors with a *deck*. The interactors in a deck are conceptually stacked on top of each other so that only the topmost interactor is visible (see Figure 12). The deck’s natural size is determined by the natural size of its largest component. A set of operations allow “shuffling” the deck to bring the desired component to the top.

Decks can be used in other contexts as well. A set of color or pattern options in a dialog box could be composed with a deck, allowing the user to flip through them until reaching the desired choice. Alternate menu entries could be stored in a deck and inserted into a menu to allow changes in the menu’s appearance without rebuilding it each time.

Single component scenes. Boxes, trays, and decks have arbitrary numbers of components. InterViews also provides several scenes that can have only one component. Such scenes are derived from the *monoscene* scene subclass and serve two purposes.

Some monoscenes serve as containers that surround another interactor. The frame used to place a border around the dialog box in the subsection “Boxes and glue” is one example. Other examples include shadow frame, which adds a drop shadow to its component, and title frame, which adds a banner. A *viewport* is a monoscene that scrolls an interactor larger than the available space. Viewports are useful for providing a scrolling interface to nonscrolling interactors.

Other monoscenes provide abstraction; they hide the internal structure of an interactor implemented as a composition. For example, the *menu* class is derived from monoscene. A menu is implemented as a box containing the interactors that represent the menu items. However, the box composition should not be visible to a programmer who wants to use the menu in a user interface. The monoscene hides the implementation of menus, making them easier to understand and allowing their structure to change without affecting other interface code.

Graphic composition

Direct manipulation editors allow the user to manipulate graphical representations of familiar objects directly. A drawing editor lets an artist draw a circle and drag it to a new location. A music editor lets a composer write music by arranging notes on staves. A schematic editor lets an engineer "wire up" graphical representations of circuits.

The programmer of such systems must provide underlying representations for the graphical objects and define the operations they perform. InterViews provides a collection of structured graphics objects that simplifies the programmer's task.

A simple drawing editor. Figure 13 depicts a drawing editor application in which the user can draw, move, and rotate rectangles and scroll and zoom the draw-

```
const int space = round(.125*inches);
TGlue* g1 = new TGlue(space, space, 0, hfil, 0, vfil);
TGlue* g2 = new TGlue(space, space, 0, hfil, 0, vfil);
/*(width, height, hshrink, hstretch, vshrink, vstretch) */

Tray* tray = new Tray(
    new HBox(
        view,
        new VBorder(1),
        new VScroller(view)
    )
);

tray -> Insert(dialog);
tray -> Align(TopLeft, dialog, g1);
tray -> Align(BottomRight, dialog, g2);
```

Figure 10. C++ code for composing the tray interface.

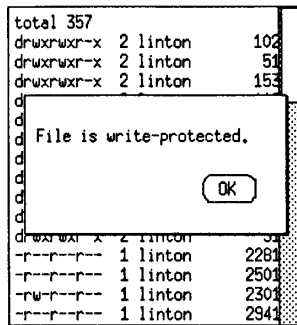


Figure 11. Tray interface after resizing.

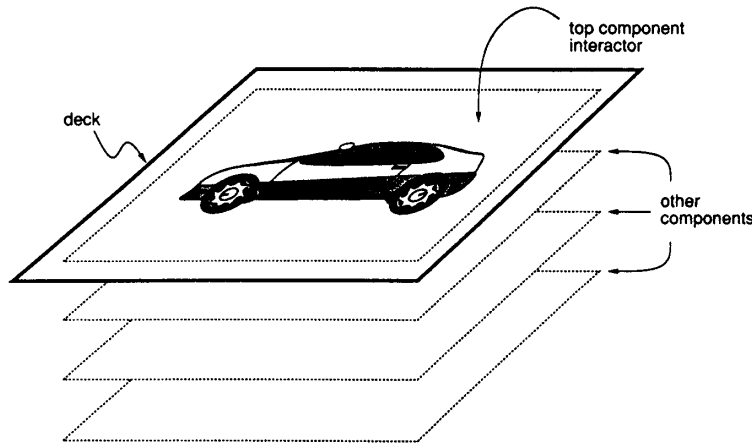


Figure 12. Composition using a deck.

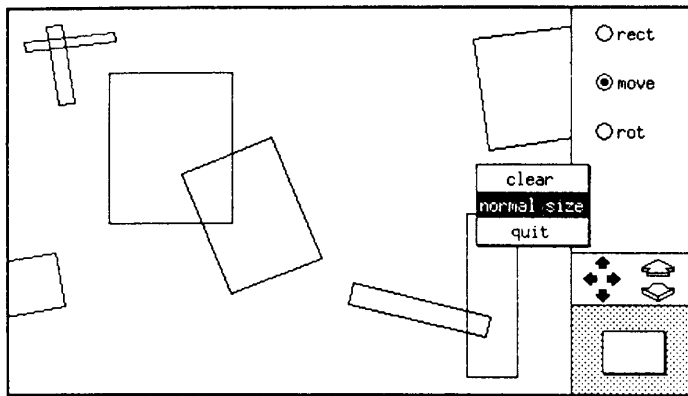


Figure 13. A simple drawing editor application.

ing area. To draw a rectangle, the user presses the “rect” button and drags out a rectangle in the drawing area. An existing rectangle can be moved or rotated by pressing the appropriate button and dragging the rectangle.

In each of these operations, the drawing editor provides animated feedback as the user creates and manipulates rectangles. Animation reinforces the user’s belief that he or she is manipulating real objects. As a rectangle is moved, for instance, its outline follows the mouse; during rotation, the outline revolves about the rectangle’s center. Such dynamic feedback is characteristic of a direct manipulation editor.

Implementing the drawing editor. A programmer can compose the elements of the user interface using InterViews interactor and graphic subclasses as shown in Figure 14. The buttons are instances of *radio button*, a predefined subclass of the button class. The interface to scrolling and zooming is provided by a *panner*, the two-dimensional scroller in the lower right of the interface. The drawing area in which the rectangles appear is a *graphic block*, an interactor that displays structured graphics objects. These elements are composed using boxes and glue. The editor’s pop-up command menu, appearing in the center-right of Figure 13, is an instance of the

menu class.

Each rectangle in the drawing is an instance of the rectangle class, a subclass of graphic. The rectangles are composed in a picture, and the picture is placed in the graphic block. The graphic block translates and scales the picture to implement scrolling and zooming. Rectangles are moved and rotated by calling transformation operations on the rectangle objects. The picture performs hit detection by returning the component that corresponds to a coordinate pair.

Semantics of graphic composition. The drawing editor demonstrates simple composition of graphics. In this example, the hierarchy of graphical objects is only one level deep; all the rectangles are children of a single parent picture. Of course, more complex hierarchies are common in a practical drawing editor. However, even the simple one-level hierarchy demonstrates the semantics of graphic composition. For example, when the graphic block applies a transformation to the picture to scroll or zoom it, the transformation affects all the rectangles in the picture. Furthermore, altering any of the picture’s graphics state attributes affects its children as well. For example, changing the picture’s brush width attribute also changes the brush widths of its children.

The composition mechanism defines how the picture’s graphics state information affects its components. A picture draws itself by drawing each component recursively with a graphics state formed by concatenating the component’s state with its own. The default semantics for concatenation are that the attributes defined by a graphic’s parent override the graphic’s own attributes. If a parent does not define a particular attribute, then the child graphic’s attribute is used. Coordinate transformations are concatenated so that the child’s transformation precedes the parent’s.

These semantics represent a kind of reverse inheritance of graphics attributes, since parents can override their children. This mechanism is useful in editors where operations performed on interior nodes of the graphic hierarchy affect the leaf graphics uniformly. Classes derived from the graphic class can redefine the semantics of concatenation if the default semantics are inappropriate.

Immediate-mode graphics. We normally do not use structured graphics objects to draw scroll bars, menus, or other user interface components that are simple to draw procedurally. Interactors use *painter* objects for this purpose. Painters provide immediate-mode drawing operations (including operations for drawing lines, filled and open shapes, and text) and operations for setting the current fill pattern, font, and other graphics state. The results of a painter drawing operation appear on the display immediately after the operation is performed. The difference between painter-generated graphics and structured graphics is that painters do not maintain state or structure that reflects what has been drawn, so there is no way to access and manipulate the graphics. In contrast, structured graphics objects maintain geometric and graphical state and can be manipulated before and after they are drawn.

Structured graphics is most appropriate where an indefinite number and variety of graphical objects are manipulated directly. It is a powerful tool for constructing graphics editors that provide an object-oriented editing metaphor because structured graphics objects embody the same metaphor. These objects typically represent the data managed by the editor. Painters should be used to draw simple, unchanging elements of the interface that do not justify the storage overhead of graphics objects.

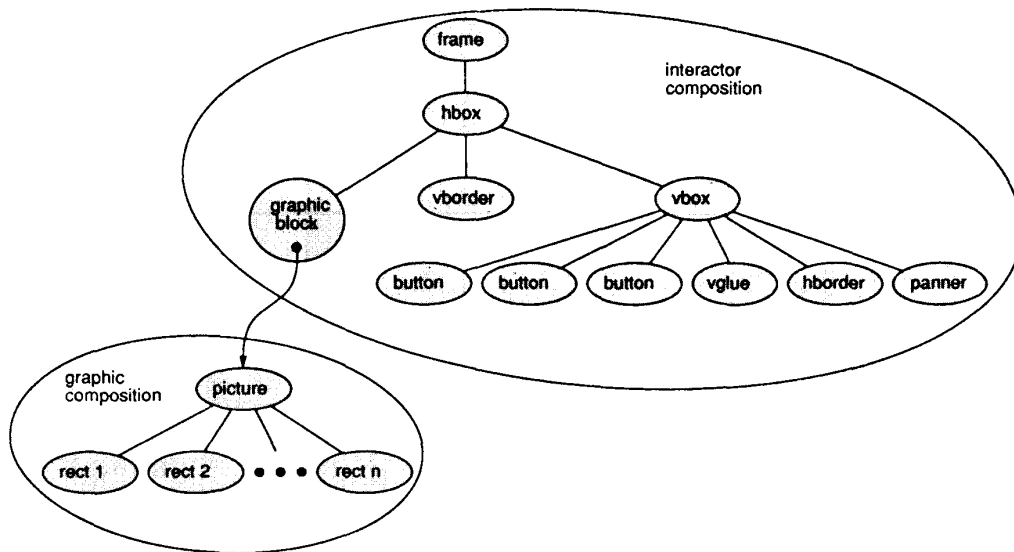


Figure 14. Drawing editor object structure.

Text composition

Direct-manipulation textual interfaces require special support to handle problems in the presentation of text, such as line and page breaking and arranging text to reflect the logical structure of a document. InterViews structured text objects simplify the implementation of direct-manipulation textual interfaces.

A simple class browser. Figure 15 shows the interface to a class browser, a simple application for perusing C++ class declarations. The browser displays a class declaration with the class name underlined and member functions in **bold**. Clicking on the class name opens a window showing documentation for the class, and clicking on a member function opens a window showing the function's definition. Text composition objects maintain the arrangement of the text. As Figure 16 shows, resizing the window reformats the text to use available space.

Implementing the class browser. Text and clause subclasses compose the text displayed in the browser. Objects of the *word* (a string of characters) and *whitespace* (blank space of a given size) classes are assembled using various composition objects so that the lines of code will fill



```

/* Base class for interactive objects. */
class Interactor {
public:
    Interactor(
        Sensor* in = stdsensor, Painter* out = stdpaint
    );
    Interactor();
    void Listen(Sensor*);
    void Iconify();
    void Read(Event&);
    virtual void Resize();
    virtual void Draw();
    virtual void Redraw(
        Coord left, Coord bottom, Coord right, Coord top
    );
    virtual void Handle(Event&);
    ...
};
  
```

Figure 15. A simple class browser application.

available space in an appropriate manner. The entire composition is placed in a *text block* (an interactor that displays structured text objects), and the text block is inserted into a frame.

Semantics of text composition. Subclasses of clause specify the way their components will be arranged. Different clauses use different strategies for using available space:

```

/* Base class for interactive objects. */
class Interactor {
public:
    Interactor(
        Sensor* in = std::sensor,
        Painter* out = std::paint
    );
    ~Interactor();
    void Listen(Sensor*);
    void Iconify();
    void Read(Event&);
    virtual void Resize();
    virtual void Draw();
    virtual void Redraw(
        Coord left,
        Coord bottom,
        Coord right,
        Coord top
    );
    virtual void Handle(Event&);
    ...
};

```

Figure 16. The class browser after resizing.

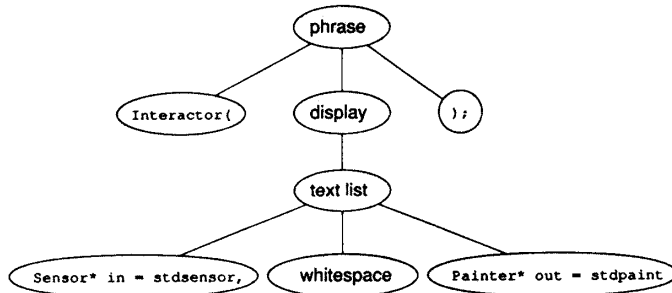


Figure 17. Object structure of the text composition for the Interactor constructor.

```

Interactor(Sensor* in = std::sensor, Painter* out = std::paint);

Interactor(
    Sensor* in = std::sensor, Painter* out = std::paint
);

Interactor(
    Sensor* in = std::sensor,
    Painter* out = std::paint
);

```

Figure 18. Possible layouts of the Interactor constructor.

- A *phrase* formats its components without regard to space. The components are simply placed end-to-end on a single line.

- A *text list* can arrange its components either horizontally or vertically. If the whole list will not fit in a horizontal format, then the list will place each component on a separate line. Text lists are used in the browser for composing the member function parameter lists.

- A *display* defines an indented layout. If the display will not fit on the current line, then it is placed on the following line with a specified indentation. The browser composes class and member function declarations using displays.

- A *sentence* will place as many components as possible on the current line and will begin a new line if necessary. The browser uses sentences for comments.

To illustrate how we can use text composition, consider the composition of the Interactor constructor in the browser (see Figure 17). The declaration is composed as a phrase with three components: the first component is a word representing the string Interactor(, the second is a display that contains a text list of the formal parameters, and the third is a word representing the string); .

Figure 18 shows that the constructor declaration will appear in one of several layouts depending on the available space. In the top example, all the text can fit on a single line. In the middle example, the available space has been reduced so that there is not enough room for the display containing the parameter list; the display is placed on a separate, indented line. In the bottom example, the available space has been reduced further, causing the text list to display vertically instead of horizontally.

Text composition is most useful when the interface requires direct manipulation of text, when the text should reflect the structural characteristics of the document, or when the text layout should automatically make good use of available space. Painters are more appropriate for embellishing interfaces with simple, noninteractive text.

Subjects and views

In InterViews we distinguish between interactive objects, which implement a user interface, and abstract objects, which encapsulate the underlying data. We refer to interactive and abstract objects as *views* and *subjects*, respectively. This separation

Making user interface development easier

We can divide software systems that facilitate construction of graphical user interfaces into two broad categories: toolkits and user-interface management systems.

Toolkits. A user interface toolkit provides programming abstractions for building user interfaces. InterViews, the X Toolkit, and the Andrew Toolkit¹ are good examples. The X Toolkit defines widget and composite classes analogous to interactors and scenes in InterViews. Tiling composites include *box* and *vpaned*, and the *form* composite allows its components to overlap. Composite objects maintain a pointer to a *geometry manager* function that is responsible for the proper layout of components. The geometry manager can be replaced at runtime to change the layout strategy.

The Andrew Toolkit includes objects that comprise the data to be edited, such as text, bitmaps, and more sophisticated objects such as spreadsheets and animation editors. Its composition mechanism allows these objects to be embedded in multimedia documents.

In addition to standard toolkit functionality, Graphical Object Workbench² allows the programmer to specify constraints between objects. Constraints can enforce dependencies between individual pieces of data. For example, the programmer can specify that a value stored in one object is a function of a value in another object. Grow also has graphical constraints for confining and connecting graphical objects. Such constraints can guarantee that a graphical object stays within a prescribed area or that two visually connected objects stay connected when one or the other is translated.

Smalltalk Model-View-Controller³ and its descendant, Apple's MacApp,⁴ are among the earliest and best-known object-oriented toolkits. MacApp differs from newer toolkits in that it implements the particular "look and feel" of Macintosh applications. MVC is unique in that it divides interface components into *model*, *view*, and *controller*. Models are similar to subjects in InterViews, controllers are responsible for input handling, and views are responsible solely for output. In contrast, other toolkits that distinguish between interactive and abstract objects put the functionality of MVC controllers and views into a single object (corresponding to an InterViews view) that handles input and output. This consolidation reflects the tight coupling between input and output in direct-manipulation interfaces. Placing responsibility for input and output in the same object reduces the total number of objects and the communication overhead between them, simplifying the toolkit and potentially increasing its efficiency.

UIMSS. User-interface management systems are generally

characterized by

- (1) complete separation of the code that implements the user interface to an application from the code for the application itself, and
- (2) support for specifying the user interface at a higher level of abstraction than general-purpose programming languages.

UIMSSs separate interface and application for some of the same reasons that many toolkits separate subjects and views, namely to isolate application code and interface specification and to allow different interfaces to the same application. However, UIMSSs do not implement any application code, whereas subjects usually do. Moreover, UIMSSs minimize the interaction between the application and the interface to maximize their independence. UIMSSs generally concentrate on abstracting the syntax and semantics of the user interface. Their main goal is to let interface designers and even end users design and modify the interface quickly without requiring extensive programming skills or knowledge of the application. To avoid conventional programming, UIMSSs use special-purpose languages or other formalisms such as finite-state transition diagrams to describe the appearance of the interface and the kinds of interaction it supports. In most UIMSSs the specification is interpreted by a runtime system incorporated into the application.

A widely known and used UIMS is Apollo's Domain/Dialog.⁵ The package consists of a compiler and a runtime library. The compiler reads a declarative description of the user interface and how it connects to the underlying application. It then generates a more compact description that is interpreted by the runtime library.

The user interface is specified in terms of interaction techniques, which correspond to primitive interface components, and structuring techniques, which are the composition mechanisms for the primitives. Domain/Dialog defines structuring techniques for arranging components into rows and columns and a "oneof" technique that displays only a single component. These structuring techniques allocate space for their components in a manner similar to InterViews' boxes and glue; they request a minimum, maximum, and optimal size from their components and distribute the available space among them.

Domain/Dialog places greater emphasis on composition than most UIMSSs, which center more on how to specify the input and output behavior of a user interface without conventional programming. Sassafra,⁶ a prototype UIMS developed at the University of Toronto, focuses on supporting concurrent user input from multiple devices and on efficient communica-

is important in many aspects of user interface design. It is a vehicle for customization, allowing programmers to present different, independently customizable interfaces to the same data. It is a useful structuring mechanism that separates user interface code from application code. It permits different representations of the

same data to be displayed simultaneously such that data changes made through one representation are immediately reflected in the others. Several other user interface packages support this separation, including the Andrew Toolkit, Smalltalk Model-View-Controller, Graphical Object Workbench, and MacApp (see the sidebar enti-

tled "Making user interface development easier").

Views in InterViews are typically implemented with compositions of interactors, graphics, and text objects. Subjects are often (but need not be) derived from the subject class. A subject maintains a list of its views. Views define an Update opera-

tion and synchronization between the modules that support user interaction. Syngraph⁷ takes a description of a user interface written in a formal grammar and generates Pascal code that implements it. Recent work by Foley et al.⁸ uses a knowledge base describing the interface to raise the level of abstraction beyond detailed assembly of components.

Another class of UIMS lets designers create a user interface by direct manipulation instead of textual specification. Research systems such as Cardelli's dialog editor⁹ and Myers' Peridot¹⁰ and commercial systems such as SmethersBarnes' Prototyper¹¹ let designers draw the user interface using a drawing editor-like metaphor. The system then generates routines that must be incorporated into the application. Cardelli's system lets designers specify the resize semantics using attachment points; an edge of a component can be attached to an arbitrary point in the interface. The component will stretch or shrink if necessary to maintain the attachment. Peridot allows the designer to specify many aspects of the interface by demonstration, inferring the proper semantics of the interface from the designer's actions. Prototyper provides a drawing editor interface to building Macintosh applications and is one of the few commercial direct-manipulation interface editors.

Toolkits, UIMSs, and InterViews. There is a growing interest in toolkits, while many researchers have begun to question whether UIMSs really help.¹² Early non-object-oriented toolkits^{13,14} were criticized as being too low-level and difficult to use, thus widening interest in UIMSs. Yet few UIMSs have gained wide acceptance. Researchers¹⁵ have identified several shortcomings of existing UIMSs:

Limited range of interfaces. Since UIMSs allow interface specification at a high level, they necessarily limit the range of interfaces they can create. This is especially true of direct-manipulation interface editors, which must rely on graphical or demonstrational specification of the interface's semantics.

Reliance on an interpreted specification language. The special-purpose language used by a traditional UIMS is likely to be unfamiliar to programmers and interface designers alike. Moreover, the language is usually inferior to established general-purpose languages, the debugging tools are primitive or nonexistent, and runtime overhead associated with interpreting the specification often degrades performance compared to conventional implementations.

Inadequacy for direct manipulation interfaces. The strict separation of application and interface code usually results in a low-bandwidth connection between the two. Thus, most UIMSs do not support interfaces requiring real-time response

to user input, such as those using rubberbanding or other animated effects.

Difficulty in adapting to change. The time needed to produce UIMSs makes it difficult to keep them in step with the latest interface designs. The problem only gets worse as interfaces become more complex.

Because InterViews is a toolkit, it avoids the problems associated with UIMSs. InterViews is distinguished from other toolkits in its variety of composition mechanisms (tiled, overlapped, stacked, constrained, and encapsulated), its support for nonlinear deformation (independent stretching and shrinking) of interactors, and its object-oriented approach to structured graphics and text. InterViews simplifies the creation of both the controlling elements of the interface (buttons and menus) and the data to be manipulated (text and graphics objects). InterViews thus offers comprehensive support for building user interfaces.

References

1. A.J. Palay et al., "The Andrew Toolkit: An Overview," *Proc. 1988 Winter USENIX*, Feb. 1988, pp. 9-21.
2. P.S. Barth, "An Object-Oriented Approach to Graphical Interfaces," *ACM Trans. Graphics*, Vol. 5, No. 2, Apr. 1986, pp. 142-172.
3. G.E. Krasner and S.T. Pope, "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80," *J. Object-Oriented Programming*, Vol. 1, No. 3, Aug.-Sept. 1988, pp. 26-49.
4. Apple Programmer's and Developer's Assoc., *MacApp: The Expandable Macintosh Application*, Renton, Wash., 1987.
5. A. Schuler et al., "ADM — A Dialog Manager," *Proc. ACM CHI 85*, Apr. 1985, pp. 177-183.
6. R.D. Hill, "Supporting Concurrency, Communication, and Synchronization in Human-Computer Interaction — The Sassafras UIMS," *ACM Trans. Graphics*, Vol. 5, No. 3, July 1986, pp. 179-210.
7. D.R. Olsen and E.P. Dempsey, "Syngraph: A Graphical User Interface Generator," *ACM SIGGraph 83*, July 1983, pp. 43-50.
8. J. Foley et al., "A Knowledge-Based User Interface Management System," *Proc. ACM CHI 88*, May 1988, pp. 67-72.
9. L. Cardelli, "Building User Interfaces by Direct Manipulation," Tech. Report 22, Digital Equipment Corp. Systems Research Center, Oct. 1987.
10. B.A. Myers, *Creating User Interfaces by Demonstration*, PhD thesis, Tech. Report CSRI-196, Computer Science Dept., Univ. of Toronto, 1987.
11. SmethersBarnes, *Prototyper Manual*, Portland, Ore., 1987.
12. J. Rosenberg et al., "UIMSs: Threat or Menace?" *Proc. ACM CHI*, May 1988, pp. 197-200.
13. Apple Computer, *Inside Macintosh, Volume 1*, Addison-Wesley, Reading, Mass., 1985.
14. Sun Microsystems, *Sun Windows Programmers' Guide*, Mountain View, Calif., 1984.
15. B.A. Myers, "Tools for Creating User Interfaces: An Introduction and Survey," Tech. Report CMU-CS-88-107, Carnegie Mellon Univ., Jan. 1988.

tion responsible for reconciling the view's appearance with the current state of the subject. Calling Notify on a subject in turn calls Update on its views, thus enabling the views to update their appearance in response to a change in the subject.

In practice it is inconvenient to force every user interface concept into the sub-

ject/view model. For example, it is unnecessary to associate a subject with every menu because interfaces seldom require multiple views of the same menu. However, many InterViews library components do use the subjects and views paradigm. Two examples relate to the implementation of scrolling and buttons.

Scrolling and perspectives. An interactor that supports scrolling and zooming maintains a *perspective*. The perspective is a subject that defines a range of coordinates representing the total extent of the interactor's output space and a subrange for the currently visible portion of the total range. For example, in the drawing editor

mentioned above, the total extent of the graphic block's perspective is obtained from the picture's bounding box; its subrange is the space the graphic block occupies on the screen. In a text editor the vertical range might be the total number of lines in a file; the subrange would be the number of lines displayed by the editor on the screen.

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 by the user manipulating one of its views.

The panner in the drawing editor is a view of the perspective associated with the editor's graphic block. The panner is really a composition of several other perspective views: a *slider*, a set of four *movers*, and two *zoomers*. Each of these elements views the same perspective; the *slider* scrolls the drawing along both the *x* and *y* axes, each mover provides incremental scrolling in one of four directions, and the *zoomers* respectively enlarge and reduce the drawing. The number of views on the same perspective is unlimited; a change made through one view of a perspective will be reflected in all its views.

The advantage of this organization is that one view of a perspective need not know about other views of the same perspective. Whenever the perspective is changed, either by the interactor or by a view, all the views are notified. Each view of the perspective is responsible for updating its appearance appropriately in response to the change. For example, when a mover or zoomer is pressed, the perspective is updated and the slider is notified automatically. The slider can then redraw itself to reflect the new perspective.

Figure 19 shows how a graphic block's perspective coordinates the scrolling operation when the user presses one of the panner's movers. The graphic block modifies its perspective on behalf of the mover because the graphic block might want to limit the amount of scrolling. In this instance, the perspective and the interactor are considered together as the subject to which views such as *panners* are attached.

Buttons and button states. The example dialog box uses a button for dismissal. In *InterViews*, a button is a view of a *button state* subject. When the user presses a button, the button sets its button state to a particular value. Several buttons can view

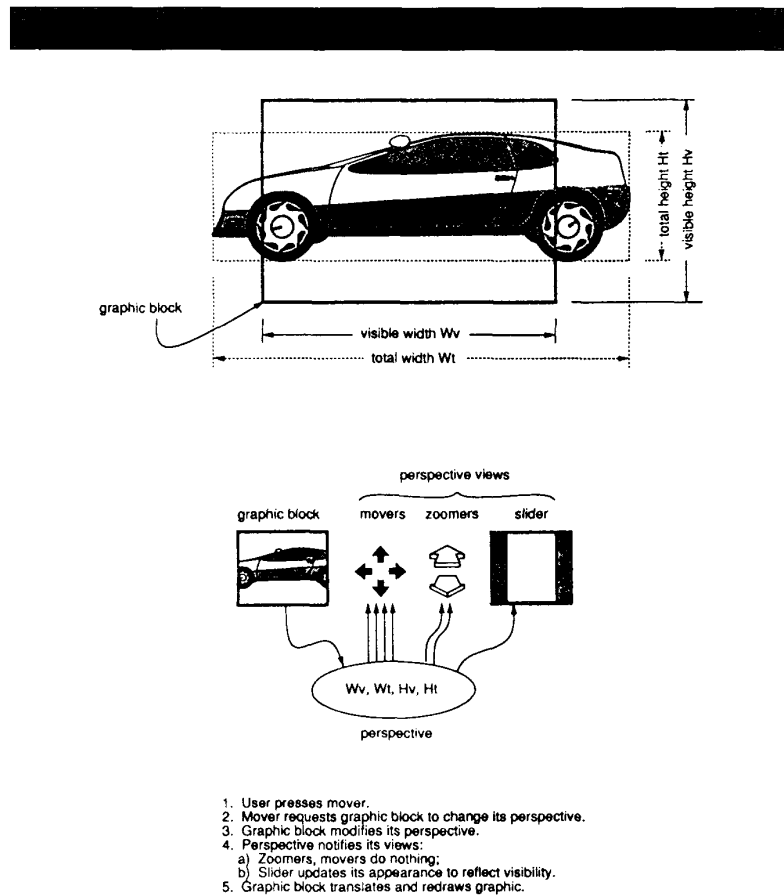


Figure 19. How a perspective coordinates scrolling of a graphic block.

a single button state; like any subject, a button state notifies all its views (buttons) when it changes.

To illustrate this, consider how *InterViews* radio buttons are implemented. A radio button acts like a tuning button on a car radio; only one button in a group of radio buttons can be "on" at a time. Radio buttons are provided when the user should select an option from several mutually exclusive choices. A single button state is the subject for a group of radio buttons. Pressing one of the radio buttons sets the button state to a particular value. The button stays pressed until the button state is changed to a different value, usually by pressing another radio button in the group.

Customization

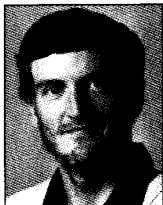
InterViews adopts the X Toolkit⁴ model to support customization of interactors. Users can define a hierarchy of attribute names and values. An interactor can retrieve the value of an attribute by name; it interprets the value to customize some aspect of its appearance or behavior. Attribute lookup involves a search through parts of the attribute hierarchy that match the interactor's position in the object instance hierarchy. Each interactor can have an instance name; interactors not explicitly named inherit a class name. The name given the interactor at the root of the instance hierarchy is usually the name of the application.

For example, suppose the application containing the example dialog box was called "hello," and the push button in the dialog box had the instance name "bye." The full name of the attribute that specifies the font for the button label would then be

```
hello.Frame.VBox.HBox.bye.font
```

Attribute names can include wild-card specifications so that one attribute can apply to several interactors. The font of the push button in the example dialog box is more likely to be specified by an attribute named `hello*PushButton.font`, which would apply to any push button in the application, or even `*font`, which would apply to any font in any application. The mechanism for accessing attributes ensures that the attribute with the most specific name is the one used to satisfy a query. The InterViews library automatically handles standard attributes such as "font" and "color."

The designer of an application chooses names for interactors that users can customize. Users specify these names to refer to interactors they want to customize. Consistency across a range of applications is achieved by a consistent choice of instance and attribute names. For example, all confirmation buttons in all "quit" dialog boxes will be red if the user lists the attribute `*quit*OK.background:red`, if all quit dialog boxes are given the instance name "quit," and if all confirmation buttons are named "OK."



Mark A. Linton is an assistant professor in the Computer Systems Laboratory of the Electrical Engineering Dept. at Stanford University. His current research interests are in programming environments, user interfaces, operating systems, and workstation architectures.

Linton received the BSE degree from Princeton University in 1978 and the MS and PhD degrees in computer science from the University of California at Berkeley in 1981 and 1983, respectively. He is the author of the Unix debugger "dbx" and the Stanford workstation benchmarks.

Current status

InterViews currently runs on Micro-VAX, Sun, Hewlett-Packard, and Apollo workstations on top of the X Window System⁵ versions 10 and 11. The library comprises roughly 30,000 lines of C++ source code, of which about 2,000 lines are X-dependent. InterViews applications do not call X routines directly and are thus isolated from the underlying window system.

We have implemented several applications on top of the library, including a scalable digital clock, a load monitor, a drawing editor, a reminder service, a window manager, and a display of incoming mail. The applications have been used daily by about 20 researchers for nearly two years, and the library is being used in development efforts at Stanford, at other universities, and in industry. We are currently using InterViews in the development of a more general drawing system, a program editor, a visual command shell, and a visual debugger.

Our experience with InterViews has convinced us of the importance of object-oriented design, subject/view separation, and composition in facilitating the implementation of user interfaces. Composition is particularly important. Providing one or two ways to combine interface elements is not enough. To really help the programmer, a user



John M. Vlissides is a research assistant in the Computer Systems Laboratory at Stanford University, where he is pursuing a PhD in electrical engineering. His dissertation work is on a generalized model for constructing graphical editing systems. His research interests also include programming environments, constraint systems, and simulation.

Vlissides earned a BS degree in electrical engineering from the University of Virginia, where he received the Electrical Engineering Dept.'s Chairperson's prize, in 1983 and an MS degree in electrical engineering from Stanford in 1985. He is a member of Eta Kappa Nu and Tau Beta Pi, and a student member of the IEEE.

interface toolkit must offer a rich set of composition mechanisms along with a variety of predefined objects. The programmer should be able to pick and choose from among the predefined components for the bulk of the interface, and the toolkit should make it easy to synthesize components unique to the application. The composition mechanisms in InterViews make this possible. □

Acknowledgments

Several people have contributed to the design and implementation of InterViews. Craig Dunwoody and Paul Hegarty participated in the design of the basic protocols. Paul also developed the window manager application, and John Interrante implemented the drawing editor. We are grateful to the growing InterViews user community for its encouragement and support. This work was funded by the Quantum project through a gift from Digital Equipment Corporation.

References

1. K. Potosnak, "Do Icons Make User Interfaces Easier to Use?," *IEEE Software*, Vol. 5, No. 3, May 1988, pp.97-99.
2. B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, Reading, Mass., 1986.
3. D. Knuth, *The TeXbook*, Addison-Wesley, Reading, Mass., 1984.
4. J. McCormack, P. Asente, and R.R. Swick, *X Toolkit Intrinsic-C Language Interface*, Digital Equipment Corp., Mar. 1988.
5. R.W. Scheifler and J. Gettys, "The X Window System," *ACM Trans. Graphics*, Vol. 5, No. 2, Apr. 1986, pp. 79-109.



Paul R. Calder is a research assistant in the Computer Systems Laboratory at Stanford University, where he is pursuing a PhD in electrical engineering. His dissertation work is on structured program editing. His research interests also include programming environments and user interfaces.

Calder received a BE(Elec) in 1978 and a MEngSc(Biomed) in 1980, both from Melbourne University, Australia. From 1980 to 1987 he served on the faculty of the Electrical Engineering Dept. at Ballarat College of Advanced Education, Australia. He is a member of the IEEE and the ACM.

Readers may contact Mark Linton at the Center for Integrated Systems, Rm. 213, Stanford University, Stanford, CA 94305.