- Glyphs provide a superset of Unidraw's structured graphics capabilities and are much less expensive. We therefore plan to retarget Unidraw to use glyphs.
- Ibuild should generate glyph code.
- Simulating glyph-based interactors would be even easier than current interactors. Most of the effort involved in writing a new interactor component is in expressing the target interactor's appearance in terms of structured graphics objects and in simulating the resizing behavior. Glyphs define both, so interactor components could use the glyphs from glyph-based interactors directly to define their appearance. This also eliminates the need to keep the builder consistent with the toolkit, though this has not proved to be a problem because the library of predefined interactors rarely changes.

Finally, we plan to extend ibuild to support standard look-and-feels and perhaps other toolkits. Already the glyph-based interactors implemented so far support three interface styles. Once ibuild supports glyphs, then, supporting multiple styles should be straightforward.

9 Conclusion

Ibuild demonstrates how a user interface builder can leverage the functionality of object-oriented libraries. It offers several features not found in existing builders:

- A composition metaphor powerful enough to define interface layout and resizing semantics without giving up a WYSIWYG interface.
- Viewing facilities at least as good as other kinds of graphics editors.
- Abstractions that go beyond the widget level into the realm of graphical object editing [11].
- An implementation that is not tied intimately to a particular widget set.
- An architecture that can support more than one toolkit, including unrealized ones.
- Extension and code integration mechanisms that leverage the C++ object model. The programmer can extend the functionality of any built-in InterViews or Unidraw object. Moreover, the programmer can change the semantics of ibuild-generated code without precluding further editing in ibuild and without special environmental support.

We see a trend towards a proliferation of object-oriented libraries that offer domain-specific solutions to application development problems. Early libraries supported basic data structures, input/output management (e.g., C++ streams), and graphical user interfaces (i.e., widgets). Newer libraries address areas like multithreading (i.e., tasking), debugging, graphical editing (e.g., Unidraw), and signal processing.

Each new library brings with it an opportunity for some sort of builder, just as widget sets begat user interface builders and Unidraw's abstractions made their way into ibuild. Builder development on this scale will be difficult without even higher-level programming abstractions, because it is unlikely that programmers will be able to exploit an instance-based implementation in builders for non-graphical libraries. This only strengthens the argument for a simulation-based approach. It also points to the need for *builder-building* abstractions that simplify builder development for any library.

References

[1] Gideon Avrahami, Kenneth P. Brooks, and Marc H. Brown. A two-view approach to constructing user interfaces. In *ACM SIGGRAPH '89 Conference Proceedings*, pages 137–146, Boston, MA, July 1989.

[2] Paul R. Calder and Mark A. Linton. Glyphs: Flyweight objects for user interfaces. In *Proceedings of the ACM SIGGRAPH Third Annual Symposium on User Interface Software and Technology*, pages 92–101, Snowbird, UT, October 1990.

[3] Luca Cardelli. Building user interfaces by direct manipulation. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software*, pages 152–166, Banff, Alberta, October 1988.

[4] Scott E. Hudson and Shamim P. Mohamed. Interactive specification of flexible user interface displays. *ACM Transactions on Information Systems*, 8(3):269–288, July 1990.

[5] T.G. Lewis, Fred T. Handloser III, Sharada Bose, and Sherry Yang. Prototypes from standard user interface management systems. *Computer*, 22(5):51–60, May 1989.

[6] Mark A. Linton, John M. Vlissides, and Paul R. Calder. Composing user interfaces with InterViews. *Computer*, 22(2):8–22, February 1989.

[7] Brad A. Myers, Brad Vander Zanden, and Roger B. Dannenberg. Creating graphical interactive application objects by demonstration. In *Proceedings of the ACM SIGGRAPH Second Annual Symposium on User Interface Software and Technology*, pages 95–104, Williamsburg, VA, November 1989.

[8] Gurminder Singh, Chun Hong Kok, and Teng Ye Ngan. Druid: A system for demonstrational rapid user interface development. In *Proceedings of the ACM SIGGRAPH Third Annual Symposium on User Interface Software and Technology*, pages 167–177, Snowbird, UT, October 1990.

[9] Stanford University. *InterViews Reference Manual, Version 3.0*, 1991.

[10] TeleSoft, Inc., San Diego, CA. *TeleUSE User's Manual*, 1990.

[11] John M. Vlissides. *Generalized Graphical Object Editing*. PhD thesis, Stanford University, 1990.

[12] John M. Vlissides and Mark A. Linton. Applying object-oriented design to structured graphics. In *Proceedings of the 1988 USENIX C++ Conference*, pages 81–94, Denver, CO, October 1988.

[13] John M. Vlissides and Mark A. Linton. Unidraw: A framework for building domain-specific graphical editors. *ACM Transactions on Information Systems*, 8(3):237–268, July 1990.

[14] Bruce F. Webster. *The NeXT Book*. Addison-Wesley, Reading, MA, 1989.

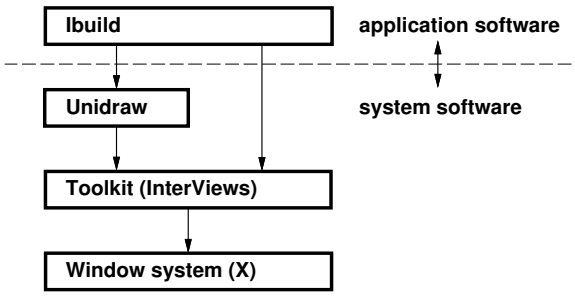

Figure 17: Layers of software underlying ibuild

The programmer can then recompile the application, and it will reflect the change in appearance. Only if the designer makes radical changes to the interface (such as deleting interactor instances upon which the application depends) should it become necessary to edit the core subclass again.

7 Implementation

Ibuild's implementation relies on Unidraw for its graphical editing capabilities and on InterViews' predefined interactors to implement its look and feel. Figure 17 depicts the dependencies between these layers of software.

7.1 Interactor Components

Ibuild defines a component subject subclass called **InteractorComp** from which to derive components that simulate the predefined InterViews interactors. InteractorComp adds two Interactor-inspired methods to the base graphical component protocol: **Reconfig** notifies the InteractorComp to compute its desired shape based on its children and whatever other criteria it considers, and **Resize** signals that the InteractorComp's canvas (actual screen space) has been allocated.

Though the Interactor base class defines considerably more protocol than these two methods, they were all that were needed for our simulation. We initially wanted ibuild to simulate input behavior, but later we realized that there was little to be gained from letting the user push buttons on and off. Therefore ibuild's interactors do not interpret input, and they and have no need for the equivalent of Interactor's Handle method or for objects that specify event interest. Rendering operations are already part of Unidraw's graphical component semantics. Child traversal, window manager, and other operations are similarly superfluous.

One benefit of these simplifications is that ibuild's interactor simulations consume only about 60% of the memory of their interactor counterparts, though this number increases with each additional view. A significant cost associated with interactors is the X window each one allocates, for which the simulated interactors have no need. Also, the multiple view feature is simply inherited from the Unidraw architecture; no special work-arounds were needed to implement it. Other features that come for free are view scrolling and zooming, incremental screen update, hit detection, Unidraw's framework for supporting direct manipulation, standard editing features such as cut/copy/paste, and arbitrary level undo and redo. Moreover, we have found it easier to simulate an interactor in Unidraw than to create the real one from scratch; thus the simulation-based implementation can serve as a testbed for interactor development.

7.2 Code Generation

CodeView is the base class from which external representation objects for each interactor component are derived. The name "CodeView" reflects the fact that external representation objects in Unidraw are simply another kind of component view. The external representation itself is InterViews and Unidraw code written in C++. A CodeView subclass instance is instantiated for the top-level interactor components at code generation time. Each subclass is responsible for generating the source code representation for the component subject it views.

Code generation is divided into multiple phases, each entailing a traversal of the entire CodeView hierarchy. This hierarchy mirrors the interactor hierarchy being simulated. Each CodeView instance is responsible for generating the proper code fragment in a given phase. For example, forward declarations are emitted in a phase in which all participating CodeViews are asked to generate their own forward declarations. Not all CodeViews participate in all phases. For example, only a few CodeView subclasses generate ButtonState declarations, because not every interactor has a ButtonState.

All the traversal machinery and protocol for emitting code are inherited from Unidraw's external representation framework. The CodeView base class initiates each pass and checks for errors. CodeView subclasses define the code fragments appropriate to each interactor component.

8 Future Directions

Ibuild has proven to be flexible and powerful enough to eliminate the need to hand-craft most interactor code and a significant amount of Unidraw code. Currently we are focusing on penetrating deeper into Unidraw abstractions. Ibuild supports only the highest-level aspects of the Unidraw architecture. It lets the user incorporate existing components, commands, tools, viewers, and editors into an interface, and the only support for extending the predefined objects' semantics is programmatically, by subclassing. Yet Unidraw defines objects at lower levels that help programmers implement graphical connectivity semantics, dataflow and dependency maintenance between components, and the direct manipulation behavior of tools. Gradually interactive analogs of these programming abstractions will make their way into ibuild's interface.

Another area we plan to address is support for InterViews' glyph abstraction [2]. Glyphs are lightweight and sharable structured graphics objects that can be used to represent the appearance of any object in a graphical user interface. Glyphs support a superset of existing interactor composition mechanisms, including support for high-quality text formatting. Traditional InterViews interactors are being replaced by glyph-based versions. This has several implications for Unidraw and ibuild:

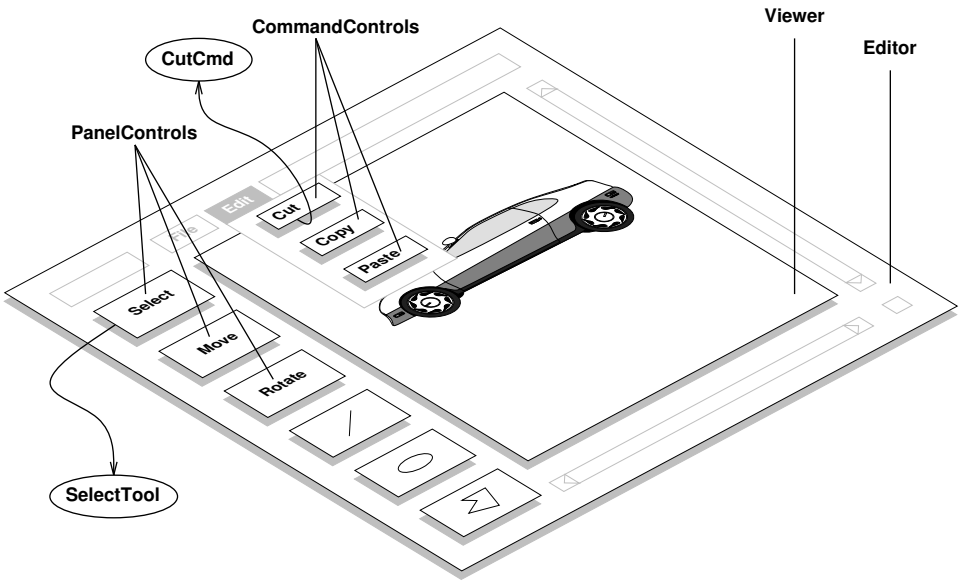

Figure 15: Composing Unidraw objects

2. An **Editor** is an interactor subclass that composes the application's interface. It associates tools (in PanelControls) and commands (in CommandControls) with one or more viewers and composes them along with other toolkit objects into a coherent user interface. A Unidraw-based application can create any number of editor objects to provide a multi-view editing environment.

Figure 15 depicts how one might compose a graphical editor with ibuild's Unidraw abstractions. PanelControls for engaging various tools appear composed in a VBox along the left side of the interface. The top three PanelControls engage tools for editing existing components, while the bottom three let the user instantiate components. CommandControls containing user-executed commands are composed in pull-down menus along the top. The Viewer in the center displays the component view hierarchy that the user edits. The interface is composed in an Editor object, which coordinates the operation of the other Unidraw-specific objects.

6 Integrating Interface and Application Code

A user interface builder's objective is to implement an interface from the user's graphical specification. To this end, most builders generate toolkit code, and ibuild is no exception. However, the user interface is normally just a part of a larger application. We have designed a novel technique for integrating interface and application code and for keeping the two consistent amidst changes to one or the other.

Ibuild generates a C++ class called the **core class** for each interactor subclass in an ibuild editing session. A user will usually define a subclass for each top-level interactor composition in the interface. The core class implements the user's interactor composition and declares all exported member names and functions.

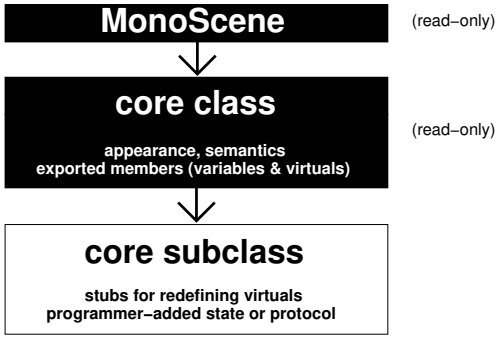

Figure 16: Core class and subclass derivation

The user exports a name by checking "Export" beside the name in the interactor information dialog. For example, ibuild generates a protected member variable for each interactor that has exported its member name, and a public virtual function may exist for each menu item in the composition. Yet protected members cannot be invoked by other objects, and the member functions have no implementation. So how do these support code integration?

The programmer could modify the core class to work with other application code, but subsequent editing and regeneration of the interface in ibuild would create a code merging problem. Instead, ibuild generates a **core subclass** in addition to the core class. As the name suggests, each core subclass is derived from a core class (Figure 16). The programmer modifies the core *subclass*, not the core class, to establish a link to the rest of the application. He can define member functions that provide controlled access to the exported member variables, and he can redefine the core class virtual functions to do useful work.

If the interface's appearance is later modified, then the user can choose to regenerate *only* the core class files, which were not modified—the changes to the core subclass are thus unaffected.

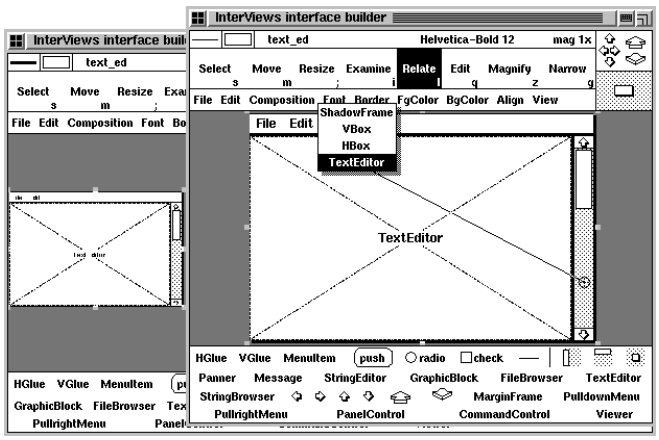

Figure 14: Specifying the target

Once the source is specified, the next step is to identify the target by clicking on the TextEditor instance (Figure 14). Again, a menu pops up to let the user choose the instance in the hierarchy. Ibuild draws a line from the source to the current mouse position to remind the user to specify the target. Upclicking the mouse over the "TextEditor" entry in the menu completes the relation. If the user now examines the scroller or either mover, the TextEditor instance's member name will appear in the information dialog's field that identifies the instance adjusted.

The user may use the Relate tool to establish a meaningful relationship between many combinations of interactors. The semantics of the relationship depends on the interactors and sometimes on the order in which the relation is made, but no more than one such relationship exists between two interactors—there can be no ambiguity. For example, relating one radio button to another ensures that they mutually exclude one another. Relating a push button to a dialog makes the push button dismiss the dialog. The Relate tool prevents the user from making meaningless relations (for example, between a pull-down menu and a border) simply by not popping up the menu (source or target) that would permit such a relation.

5 Unidraw Abstractions

In addition to providing widget-level support, ibuild also incorporates Unidraw abstractions that simplify the construction of graphical editors. Unidraw is designed to span the gap between traditional user interface toolkits and the implementation requirements of these applications. It partitions their basic functionality into four class hierarchies:

1. **Components** represent the elements in a graphical editing domain, for example, geometric shapes in technical drawing, schematics of electronic parts in circuit layout, and notes in written music. Components encapsulate the appearance and semantics of these elements. The user arranges components to convey information in the domain of interest. A component is made up of a **subject** and zero or more **views**: the subject defines the component's semantics, while views define presentations of the subject. Components can be structured hierarchically.

2. **Tools** support direct manipulation of components. Tools employ animation and other visual effects for immediate feedback to reinforce the user's perception that he is dealing with real objects. Examples include tools for selecting components for subsequent editing, for applying coordinate transformations such as translation and rotation, and for connecting components.

3. **Commands** define operations on components and other objects. Commands are similar to messages in traditional object-oriented systems in that components can receive and respond to them. Commands can also be executed in isolation to perform arbitrary computation, and they can reverse the effects of such execution to support undo. Examples include commands for changing the attributes of a component, duplicating a component, and grouping several components into a composite component.

4. **External representations** convey domain-specific information outside the editor. Each component can define one or more external representations of itself. For example, a transistor component can define both a PostScript representation for printing and a netlist representation for circuit simulation; each is generated by a different class of external representation. An external representation object thus defines a one-way mapping between a component and its representation in an outside format.

The Unidraw library defines the base classes and a set of subclasses for each of these elements. The predefined subclasses support basic drawing editing capabilities. For example, the library includes:

- components for elementary shapes such as lines and polygons
- tools for creating and selecting components and for applying coordinate transformations by direct manipulation
- commands for cut, copy, paste, undo, redo, changing component attributes, saving and restoring drawings
- external representation objects for generating PostScript

The library also defines special interactors to engage and control these abstractions. It defines two subclasses of InterViews' **Control** class, which is the base class for interactors that execute an action: **CommandControl** provides an interface for executing commands from menus, and **PanelControl** engages a particular tool. Ibuild includes these interactor classes and lets the user specify via the Examine tool the command or tool subclass they use. A user can specify either a predefined command or tool subclass by name; he can also extend the existing classes programmatically by subclassing them as he can interactors.

Ibuild currently supports two additional Unidraw classes:

1. A **Viewer** displays a graphical component view, most often the root view in a hierarchy. A viewer provides a framework for displaying the view, supporting such "non-semantic" manipulations as scrolling and zooming. Viewers also take raw window system or toolkit events and translate them into Unidraw protocol requests.

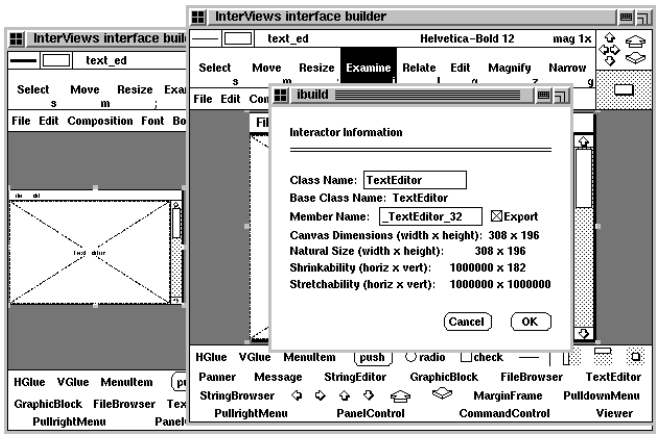

Figure 12: TextEditor attributes

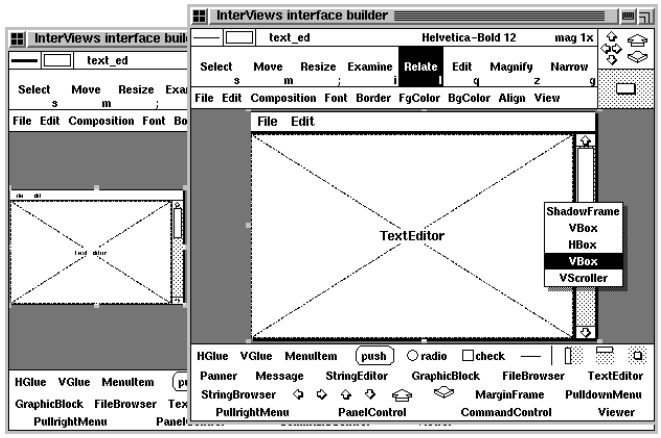

Figure 13: Specifying the source with the Relate tool

Info... choice) or the static customization properties (via **Props...**). In either case, a submenu analogous to the Narrow tool's lets the user choose the instance to examine in the composition, from the leaf instance clicked upon to the topmost interactor in the hierarchy.

If the user chooses to examine the TextEditor's run-time information, then an **Interactor Information** dialog will appear as shown in Figure 12. Had the user chosen to examine the TextEditor's properties, an **Interactor Properties** dialog would have appeared in which to type static attribute names and their values. The Interactor Properties dialog is the same for all interactors: it has a simple text editor-based interface for specifying the properties. The Interactor Information dialog varies from interactor to interactor, though all have a class name, a base class name, a member name, canvas dimensions, and shape information.

When the class name and base class name are the same, ibuild will instantiate the corresponding class. But by changing the class name, the user directs ibuild to declare a new subclass of the base class and to use an instance of that subclass. The generated code will include a subclass declaration that the user can modify to specialize the behavior of the base class. For example, our text editing interface would probably define and use a subclass of TextEditor (instead of TextEditor itself) that implements application-specific input handling. Specifying subclasses provides a simple way to extend the behavior of ibuild's predefined interactors programmatically.

The interactor's member name identifies the interactor to application code should the application need to access it; this issue is discussed further in Section 6. The interactor's canvas defines the actual screen space the interactor occupies, which may be different from the desired size specified by its shape. Most interactors computer their shape from information they keep internally; thus their shape information is not editable in the information dialog. However, glue and other interactors that parameterize their shape have editable shape information fields.

TextEditors have a minimum of run-time attributes. Below is a sampling of the information that can be specified for different interactors through the Examine tool interface:

- ScrollBars and Movers have a field for specifying the member name of the interactor they scroll.
- Menus have a field for specifying a member function to call when invoked.
- Buttons provide an interface for specifying ButtonState information [6] along with a member function to call whenever the ButtonState's value changes.
- FileBrowser (a scrollable list of files in a directory) provides fields for specifying a directory to search and regular expressions for filtering files.
- The pop-up menu for GraphicBlock (an interactor that displays, scrolls, and zooms structured graphics objects [12]) and Viewer (a Unidraw object derived from GraphicBlock) adds a **Graphics** entry that lets the user specify the graphics they display through a drawing editor interface (idraw [9]).

4.2 Relating Interactors

Some interface semantics establish relationships between interactors. For example, a scroll bar normally scrolls another interactor, some button choices might exclude others, and double-clicking in a scrollable list of choices might be equivalent to pressing a button to dismiss the dialog. These relationships can be established through dialog boxes produced by the Examine tool, but such interaction is indirect at best and clumsy in practice.

Ibuild's **Relate** tool provides a direct-manipulation interface for establishing relationships between interactors. The Relate tool involves two interactor instances, the **source** and the **target**. The user identifies the source and then the target by clicking on each in turn. As with the Examine tool, the Relate tool pops up a menu in response to each downclick to let the user select the instance in the hierarchy being related.

To demonstrate how the Relate tool works, we will use it in our text editing example to make the scroller and movers adjust the TextEditor instance. Figure 13 shows the pop-up menu produced by clicking on the VScroller instance. The user specifies the source by selecting an entry in the pop-up menu. In this case the user could choose the VScroller, but then the movers must be related separately. Instead, the user can relate both movers and the scroller at once by making the VBox that composes them the source, thereby specifying a recursive relation.

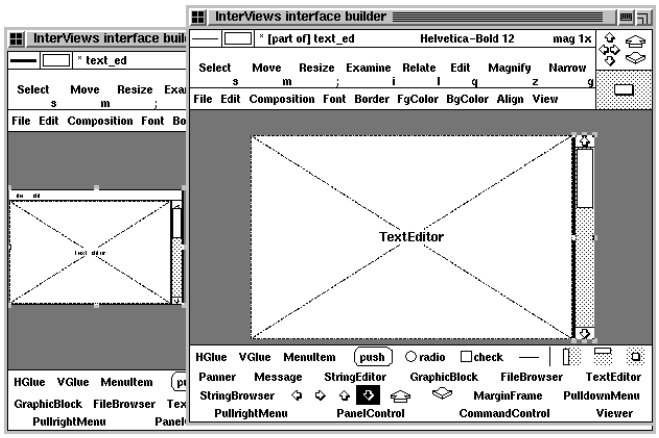

Figure 8: Recomposed interface

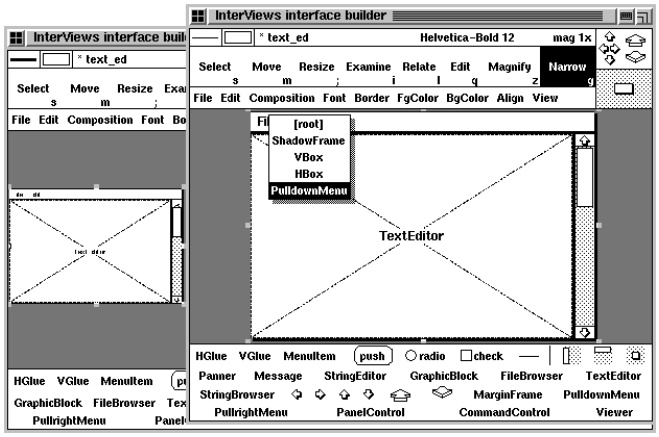

Figure 9: Narrowing in on a pull-down menu

VScroller. These additions appear in the original view as well. Ibuild does not, however, enforce all composition semantics constantly when editing a subview. It is important to preserve the user's freedom to position interactors in the most natural way before composing them. Restricting an interactor's placement prior to being composed is unnecessary because composition ultimately determines an interactor's placement. Avoiding any such restriction lets the user arrange interactors freely, giving him a feel for spatial relationships *before* committing them to a particular composition. Ibuild provides a **Natural Size** command that repositions the selected compositions to reflect their natural appearance. Figure 8 shows the interface at its natural size.

The user employs the same narrowing mechanism to specify the body of a menu. Figure 9 shows the user narrowing in on the "Edit" pull-down menu with the Narrow tool. Inside the pull-down menu the user sees the tag in its highlighted state, suggesting that the menu's body is exposed. The user may then instantiate the interactors that make up the body, including other menus or compositions of interactors (Figure 10). As in other subviews, the user may place newly instantiated interactors anywhere; the menu's composition semantics will define its children's ultimate

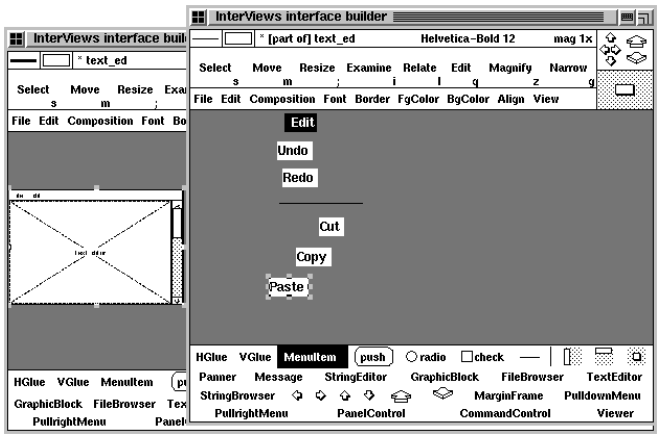

Figure 10: Editing a pull-down menu's body

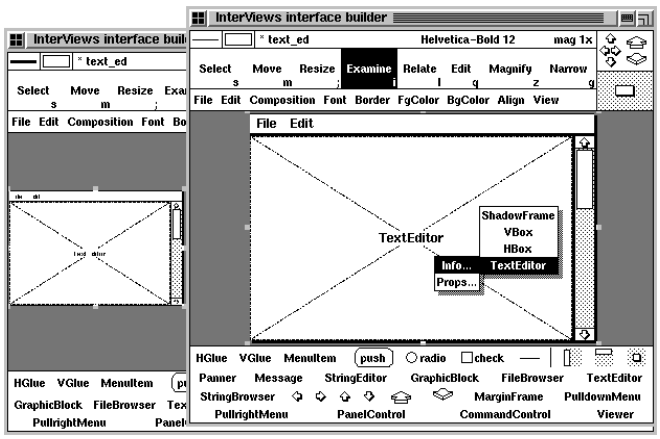

Figure 11: Examining a TextEditor

positions. In this case, the pull-down menu will tile its body's children vertically.

The ability to edit parts of an interactor composition is an indispensable one in any builder that supports hierarchy. Note that the composition model that InterViews defines and ibuild inherits at first glance seems an inherently bottom-up approach to interface design. Yet with the subview capability, the designer is free to revisit and expand parts of the composition in a top-down manner. Menu specification in particular is top-down.

4 Interactor Attributes

In addition to a shape, interactors have other attributes of interest to the user. Ibuild provides an **Examine** tool for inspecting and modifying these attributes.

4.1 Examining an Interactor

Figure 11 shows the two-level pop-up menu produced by the Examine tool when the user clicks upon the TextEditor instance in our example interface. In this case the user has the option of examining either run-time information about the interactor (via the

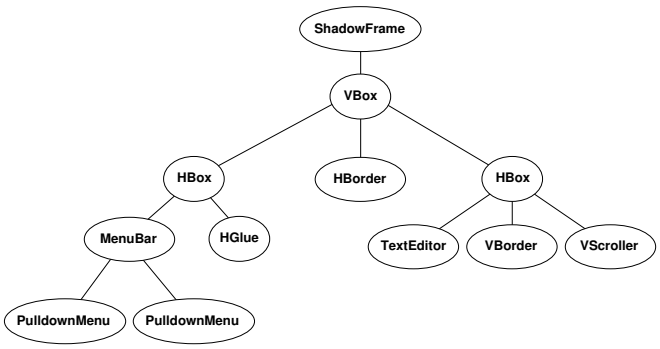

Figure 5: Interactor hierarchy for the text editing interface

both empty space and slack between interactors. Glue normally deforms sooner than other, non-trivial interactors when there is more or less space than the natural amount. In this case, the glue in the HBox effectively left-justifies its MenuBar sibling.

The MenuBar contains two **PulldownMenu** instances, which it tiles horizontally. It also establishes the dependencies between the menus so that they work in concert. The only visible manifestation of the pull-down menus in the MenuBar is the tag that the user clicks on to expose the menu's **body**, which contains its choices. A pull-down menu tiles the interactors in its body vertically. Any interactor can appear in a menu's body, including other menus, thereby supporting hierarchical menus. We explain how the body is composed in the next section.

Of existing builders, FormsVBT's composition model is closest to ibuild's. There are significant differences, however. FormsVBT interactors cannot define shrinkability, thus limiting the range of resize behaviors. Its composition objects have visible manifestations that do not appear in the final interface. Moreover, Ibuild supports a greater variety of composition mechanisms, including nearly all that InterViews defines [6].

3.2 Modifying the Composition

How does one modify part of the composition without disturbing other parts? Stated another way, how does one refer to and edit composition objects buried deep in the interface? One could start from scratch or could dissolve the compositions to expose the leaf interactors. But both of these alternatives force the user to repeat work already performed, which is undesirable even for small interfaces.

Like TeleUSE, ibuild uses multiple views to address this problem, but ibuild's views are always WYSIWYG. Suppose we want to augment our text editing application's scrolling interface with two buttons that support incremental vertical scrolling, either up or down. In InterViews terminology we could place an **UpMover** above the VScroller and a **DownMover** below it. This calls for a composition in which the movers and the scroller are tiled vertically. We therefore introduce a VBox into the existing interface.

First we will create a separate view of the interface in which to work, keeping the original view. Any changes we make in one view will appear synchronously in all other views. Each view can be scrolled and zoomed independently, and interactors may be selected in multiple views independently.

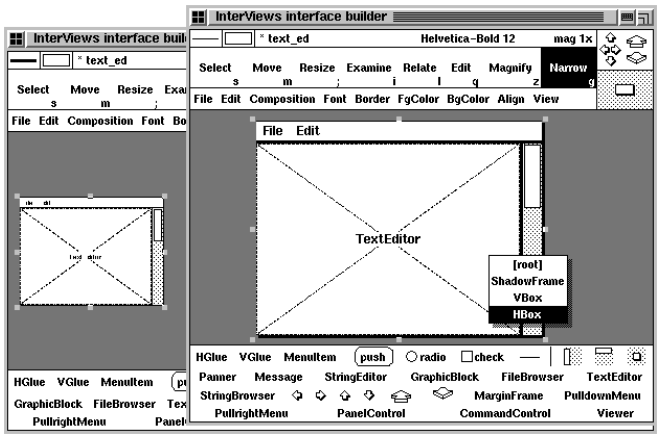

Figure 6: Narrowing a separate view

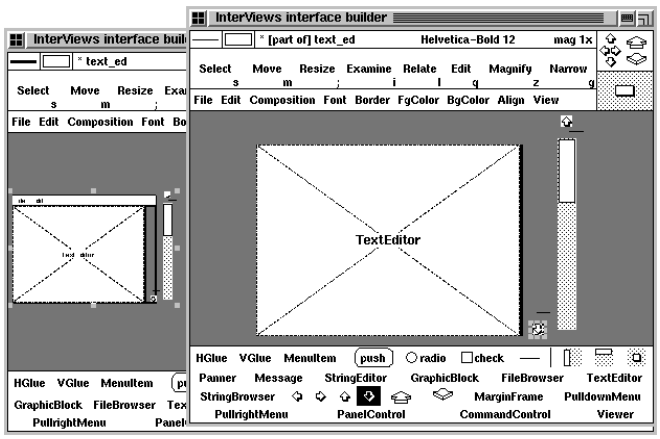

Figure 7: Editing inside the HBox

Initially both views display the interface in its entirety. Now we will **narrow** the second view into the HBox composition containing the TextEditor, the VBorder, and the VScroller. In Figure 6 the user has clicked on the VScroller with ibuild's **Narrow** tool engaged. Doing so produces a pop-up menu containing entries that reflect the instance hierarchy in the scroller's subtree. Each entry is the class name of an instance along the path from the root of the hierarchy (the topmost entry in the menu) to the parent of the instance clicked upon (the bottommost menu entry, in this case, the HBox containing the VScroller). Selecting one of these entries will narrow the view to the chosen node, eliding other parts of the composition. We can then edit that portion of the interface out of context, as if it was the entire interface.

Figure 7 depicts the two views of the interface with the one on the right narrowed to view the HBox's contents. Note that the original view (on the left) merely provides context by displaying the overall interface. The user may create as many or as few views as desired, and each may be narrowed independently.

The HBox's siblings and ancestors are elided in the narrowed view, and its children can be selected individually. The user has added an UpMover, a DownMover, and two HBorders to the interface and is about to compose them in a VBox along with the

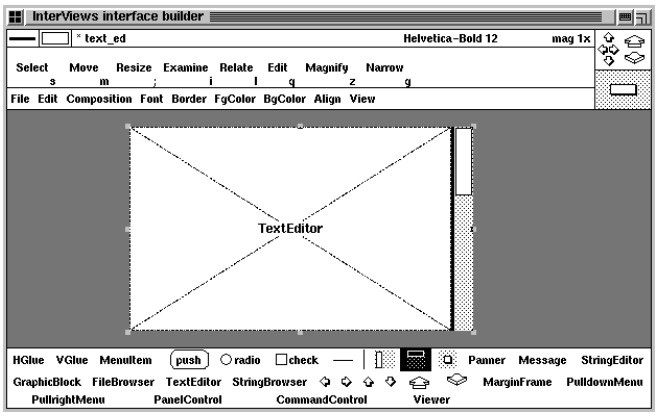

Figure 2: TextEditor, VBorder, and VScroller instances composed in an HBox

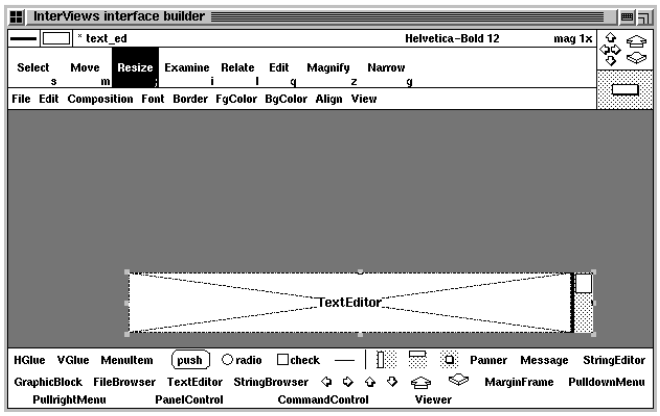

Figure 3: HBox composition after resizing

- A **TextEditor** instance provides a multiline text editing interface but defines no input handling.
- A **VBorder** defines a vertical border.
- A **VScroller** supplies an interface for scrolling in the vertical dimension.

Each object represents an instance of an Interactor subclass. The user can instantiate any of the interactors that InterViews predefines by clicking in the viewing area with the appropriate **creation tool** engaged. The creation tools installed currently appear along the bottom of ibuild's interface. The manipulation semantics for instantiating an object vary with the interactor. For example, the TextEditor instance is created by sweeping out a rectangular area of the desired size, while the VBorder is specified by dragging a vertical line.

All interactors have a **shape**. The shape defines a natural size (width and height) for the interactor as well as the amount by which the interactor is prepared to stretch or shrink from the natural size. For example, the natural size of both the VBorder and VScroller is zero in the vertical dimension, and both can stretch "infinitely" in this dimension. The TextEditor is infinitely stretchable and shrinkable as well, but its natural size is defined implicitly when it is created by direct manipulation.

Each object in the interface is thus far uncomposed and independent. Now we will compose the interactors in an **HBox** so that they tile horizontally. In general, composing interactors in ibuild involves selecting the instances to be composed and then choosing a composition mechanism from the **Composition** menu, for example, "HBox." Once composed, the interactors are no longer independent; in this case the HBox instance becomes the only selectable component in this view of the interface (see Figure 2). Note that the border and scroller have stretched vertically to accommodate the TextEditor's natural height. This follows from the HBox's shape semantics, which specify that the natural height of an HBox is the maximum of the natural heights of its children.

We can resize the interface as it now stands with ibuild's **Resize** tool (Figure 3), which will simulate a window manager's resizing operation. Doing so can give a designer the feedback needed to ensure a pleasing layout in the face of resizes. Note

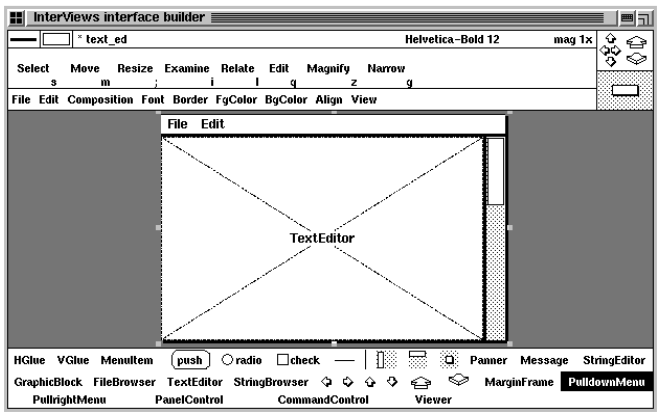

Figure 4: Text editing interface, fully composed

that the resize semantics are a natural outgrowth of the composition mechanisms—no graphical notation apart from the interface itself is required to specify the layout constraints, and the builder's rendition of the interface remains WYSIWYG. To ibuild's user, the objects that define layout and resizing semantics are an integral part of the interface. The end-user, on the other hand, has no knowledge of their presence.

The HBox object can now participate in a larger composition that includes pull-down menus. Figure 4 shows the text editing interface once it is fully composed, and Figure 5 depicts its interactor hierarchy schematically. The top-level interactor in the interface is a **ShadowFrame**, a composition that draws a drop shadow around its (one and only) component. The ShadowFrame's component is a **VBox**, which tiles its components vertically. Inside the VBox is the HBox composed earlier, a horizontal border (**HBorder**), and another HBox containing a **MenuBar** object and a piece of horizontal glue, or **HGlue**. HGlue (and its vertical counterpart **VGlue**) are interactor subclasses that define whitespace of user-specified natural size, stretchability, and shrinkability along their major axis (which for HGlue is the horizontal axis); they have zero natural size and infinite stretchability in their minor axis. Glue is composed along with other elements of the interface to provide

1. Most support the interactive layout of a user interface, that is, absolute positioning of widgets by direct manipulation.
2. A few offer mechanisms for expressing spatial relationships between widgets, which define the **resize semantics** of an interface.
3. Some support hierarchical structuring of widgets to aid in the specification of complex layouts.
4. Some research builders offer demonstration-based specification. At least one research builder, Lapidary [7], supports demonstrational *widget* specification.

A common way to support specification of resize semantics is through a graphical notation. In Cardelli's DialogEditor [3], one of the earliest builders, the user draws **attachment points** that keep an edge of a widget attached to an arbitrary point in the interface. The component will stretch or shrink if necessary to maintain the attachment. OPUS [4] extends this model to include a variety of notations for aligning widgets and spacing them proportionately. The main drawback of these notations, apart from their often-cryptic appearance, is that their presence interferes with the interface being built, thereby mitigating the benefits of a WYSIWYG editor. NeXT's Interface Builder [14] takes a simpler approach. The user can introduce integral numbers of **springs** around widgets that define how readily the widget deforms when pressed or pulled by the window it occupies. The springs do not appear in the interface but are specified instead through dialog boxes. This scheme is simple and non-intrusive but cannot express many common layouts, such as mutually centered or tiled widget arrangements. Avrahami, et al.'s FormsVBT dialog builder [1] employs a TEX-based boxes-and-glue model for widget layout similar to InterViews' (and ibuild's). While FormsVBT's direct-manipulation interface is not WYSIWYG, the system can also display a non-editable WYSIWYG view.

The more elements a user interface has, the more important it is for the builder to support hierarchy. The reason comparatively few builders support hierarchy stems from their WYSIWYG nature: the widget hierarchy has no visible manifestation in the interface from the user's perspective. Thus hierarchical builders either depart from a truly WYSIWYG model or they introduce additional, hierarchical views of the interface, or both. For example, child widget compositions in OPUS are elided as gray boxes. The user can expand a box into a window that contains the child's contents with its own children elided. Navigating through a deeply nested hierarchy can be cumbersome, however, since it is not possible to jump more than one level at a time. TeleSoft's TeleUSE [10] provides a tree view of the widget hierarchy and allows limited editing to take place there. FormsVBT provides a textual view for viewing and specifying the interface in an s-expression-based language. Unfortunately, it is often difficult for a user to map between widgets in the graphical view and nodes in a tree or s-expressions in a program.

Another approach to visual interface specification is through demonstrational techniques. These have been applied most successfully to solving the layout problem, as in OSU [5], Lapidary, and Druid [8], though each of these systems lets the user specify certain interface semantics by demonstration. The known problems with demonstrational specification in general apply equally well to its use in interface builders:

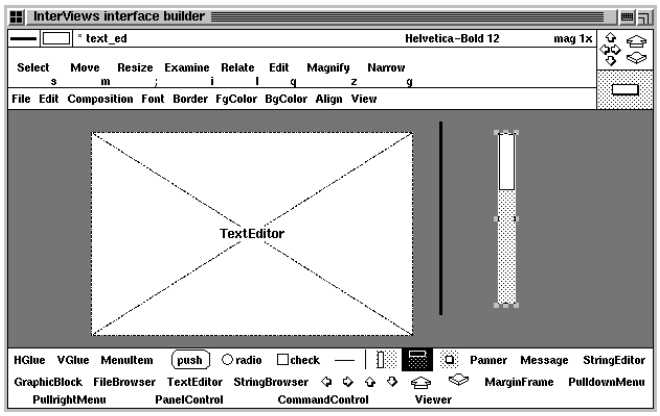

Figure 1: Uncomposed TextEditor, VBorder, and VScroller instances

- Deducing non-trivial interface semantics from a limited number of user actions is difficult.
- The inferred behavior is not always predictable.
- There is usually no way to edit a demonstration.
- The system understands a limited number of demonstrations and must have a fallback specification mechanism when these are insufficient to express desired behavior.
- It is difficult to extend the system to understand new demonstrations.

There are other issues that current builders fail to address adequately. No current builder eliminates the need for conventional programming in application development. Therefore a programmer must at some point integrate builder-generated code with the rest of the application. To carry out the integration, the programmer usually needs to access or even modify objects in the generated code. Once generated code is modified, however, the programmer can no longer edit the interface in the builder without merging his changes into code generated subsequently. Code consistency thus becomes a problem. A related problem for builders is providing a mechanism for extending the functionality of the builder's built-in widgets. Finally, no current builder has proven practical for specifying interfaces beyond the widget level. For example, assembling a non-trivial, production-quality graphical editing application is beyond the capabilities of current systems.

3 Composing an Interface with Ibuild

We will introduce ibuild's interface and capabilities by composing a simple text editing interface.

3.1 Basic Composition

Figure 1 shows the interface being built with ibuild. Three objects are instantiated initially (from left to right):

A Unidraw-Based User Interface Builder

John M. Vlissides and Steven Tang
Center for Integrated Systems
Stanford University
Stanford, California 94305

Abstract

Ibuild is a user interface builder that lets a user manipulate simulations of toolkit objects rather than actual toolkit objects. Ibuild is built with Unidraw, a framework for building graphical editors that is part of the InterViews toolkit. Unidraw makes the simulation-based approach attractive. Simulating toolkit objects in Unidraw makes it easier to support editing facilities that are common in other kinds of graphical editors, and it keeps the builder insulated from a particular toolkit implementation. Ibuild supports direct manipulation analogs of InterViews' composition mechanisms, which simplify the specification of an interface's layout and resize semantics. Ibuild also leverages the C++ inheritance mechanism to decouple builder-generated code from the rest of the application. And while current user interface builders stop at the widget level, ibuild incorporates Unidraw abstractions to simplify the implementation of graphical editors.

Keywords: user interface builders, user interface toolkits, direct manipulation, graphical constraints

1 Introduction

Few user interface builders today are written from scratch. Like other interactive applications, most benefit from a toolkit-based implementation. But in addition to using a toolkit in the normal way, most builders also use instances of toolkit classes as data objects: a user manipulates these instances directly in the builder to construct a user interface. Such **instance-based** builders employ mechanisms that keep the toolkit objects passive as the designer assembles the interface. This approach has certain advantages: the builder's objects look and feel exactly like the toolkit objects do, since they are one in the same, and testing the final interface involves simply disabling the mechanism that keeps the objects passive during the building process.

But the instance-based approach has several disadvantages as well. To implement passivation, the builder must subvert normal toolkit semantics, which usually requires knowledge of toolkit internals. The builder and toolkit implementations are thus closely coupled, making it harder to retarget the builder to a new toolkit. Moreover, an instance-based approach complicates the implementation of features that are taken for granted in graphical editors for other domains, features such as scrolling and zooming the interface, eliding parts of it, and supporting multiple views.

Ibuild is a user interface builder for the InterViews toolkit [6] that lets a user manipulate *simulations* of toolkit objects. Ibuild's implementation of these objects is independent of the toolkit's implementation. Builder developers have avoided simulation-based approaches because they offer no implementation advantage over an instance-based approach when the builder is developed on top of a traditional user interface toolkit. Ibuild, however, takes advantage of Unidraw [13], a framework for building direct manipulation graphical editors. Unidraw provides abstractions above the toolkit level that simplify the construction of such applications. Unidraw makes a simulation-based approach attractive, and it has enabled us to explore the advantages of simulation as an alternative to the instance-based approach. Moreover, Unidraw provides the basis for builder abstractions beyond the widget level.

In addition to standard toolkit objects such as scroll bars, buttons, and menus (generally known as **widgets** or, in InterViews terminology, **interactors**), ibuild provides direct-manipulation analogs of InterViews composition mechanisms for specifying spatial relationships between interactors. Simulation makes it possible to let the user modify part of the resulting hierarchical interactor structure without disturbing unrelated parts and without compromising the builder's what-you-see-is-what-you-get (WYSIWYG) model. The user can also establish relationships between interactors by direct manipulation and can incorporate Unidraw abstractions into the interface to support graphical editing applications. Ibuild generates InterViews code that implements the interface, and it uses the C++ inheritance mechanism to insulate ibuild-generated code from application code and to permit extension of built-in interactors.

This paper describes ibuild's design and implementation on top of Unidraw.

2 Background

User interface builders promote visual user interface design and development to minimize the need for conventional programming. Current commercial and research builders support interface specification on several levels:

This research has been supported by a grant from Fujitsu America, Inc., by the SRC under contract 90-MC-106, by the U.S. Navy/DARPA under contract N00014-87-K-0729, by the NASA CASIS project under Contract NAGW 419, by the Quantum project through a gift from Digital Equipment Corporation, and by a grant from the Charles Lee Powell foundation.