Ibuild User's Guide

September 30, 1992

John Vlissides, IBM T.J. Watson Research
Steve Tang, Stanford University
Charles Brauer, Fujitsu Network Transmission Systems, Inc.

Foreward

In the spring of 1990, John Vlissides completed his doctoral thesis on Unidraw, an object-oriented architecture for building interactive graphics applications he calls ‘‘graphical object editors.’’ Such applications provide a direct manipulation metaphor for defining and arranging graphical objects to convey information in some domain of interest. The centerpiece of John’s work is a library of C++ classes that implement the Unidraw architecture.

To demonstrate Unidraw’s effectiveness, John used the library to build three experimental domain-specific graphical object editors: a drawing editor similar to MacDraw, a schematic capture system for circuit design, and a user interface builder. Since then a colleague of John’s, Steve Tang, has refined and expanded the experimental user interface builder into the application described in this document.

None of this work would have been possible if it were not for Professor Mark Linton. The InterViews system, of which Unidraw is a part, was developed under Mark’s guidance, and he continues to devote his energies to its development and to establishing it as an international standard. Without InterViews, our group at Fujitsu would still be programming in C and struggling with disgusting C-based toolkits.

Charles Brauer
Fujitsu America

Table of Contents

Chapter 1 Introduction

This document describes ibuild, a user interface builder for the InterViews toolkit. Ibuild provides an interactive environment for creating graphical user interfaces to workstation applications. It lets a designer specify many aspects of the interface through a drawing editor-like metaphor, and it can generate source code that implements the interface from the designer's graphical specification. A programmer can then integrate this code with the rest of the application.

Ibuild is designed to increase productivity by eliminating the tedious aspects of assembling user interface components, which normally involve textual programming and repeated edit-compile debug cycles. With ibuild, the designer is free to concentrate on the aesthetics and behavior of the user interface without worrying about programming language and toolkit details.

Using this Guide

We assume you understand object-oriented programming concepts (e.g., classes, instances, inheritance) and the fundamentals of InterViews (e.g., interactors, boxes and glue). Each chapter presents a basic part of ibuild's functionality by guiding you through step-by-step examples:

- Chapter 2 describes ibuild's basic operation and demonstrates how to build and test a simple interface.
- Chapter 3 shows you how to compose sophisticated interfaces out of simpler ones and fine-tune their appearance.
- Chapter 4 describes how to connect the interface that ibuild generates to the underlying application.
- Chapter 5 introduces some of ibuild's advanced features, including how to create and use multiple views of an interface and how to supplement ibuild's set of predefined interface elements with ones of your own design.
- Chapter 6 introduces structured graphics with a histogram example.
- Chapter 7 illustrates an extended example that uses ibuild to build the interface to a simple text editor application.
- The Appendix describes shortcuts for common operations.

Chapter 2 The Basics

In this chapter you'll use ibuild to construct a simple interface called **Hello** that displays the string "Hello world" in a window. But first we'll describe ibuild's interface so that you can identify its parts when we refer to them.

Ibuild's Interface

Figure 1 shows how ibuild looks when you start it for the first time and labels its different parts. Interfaces are assembled in the **viewing area**, the large rectangular region in the center. You can scroll and zoom the viewing area to get a better view with the **panner** in the upper right-hand corner. The viewing area simulates the screen on which an interface will appear. But while a real screen has finite size, the viewing area is infinite, giving you plenty of room to work with the interface. The viewing area displays a visual boundary that depicts the real screen to give you a feel for how much screen space an interface occupies.

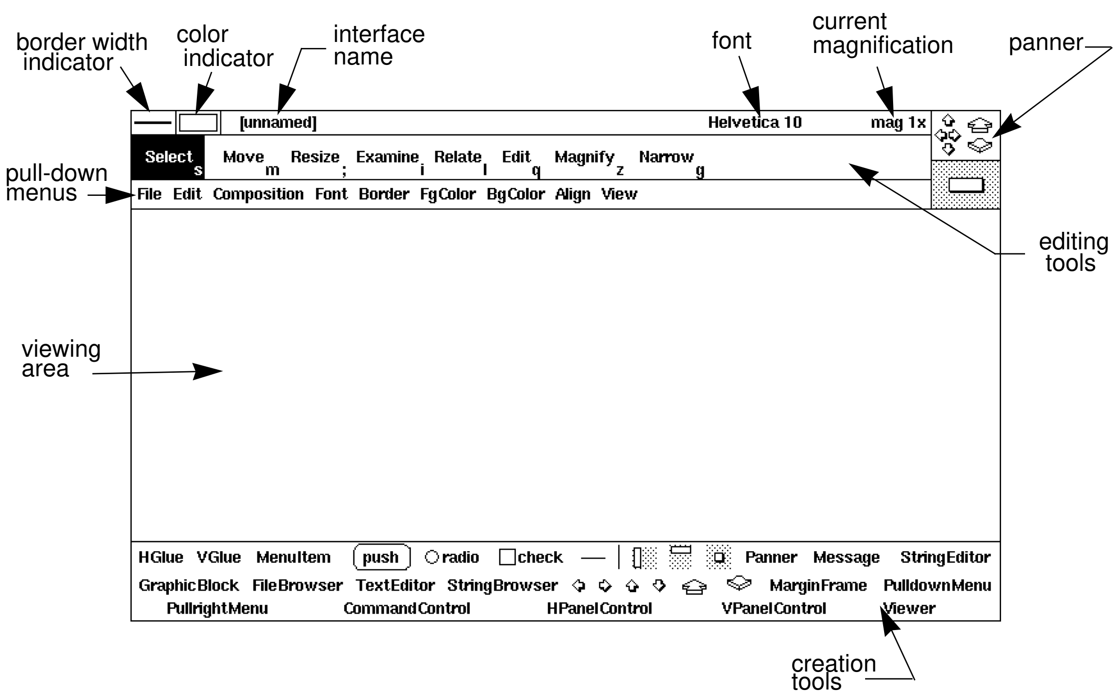

Figure 1.

Directly below the viewing area is a palette of **creation tools** that let you create instances of interface elements (interactors) in the viewing area. Ibuild provides a different creation tool for each kind of interactor: one for creating push buttons, one for menu items, and so on. To create an interactor instance, simply click on its creation tool to engage that tool and then click in the viewing area wherever you want the instance to appear. You can create as many instances as you like by clicking repeatedly in the viewing area.

Above the viewing area are **pull-down menus** containing commands for editing different aspects of the interface, for saving it and retrieving interfaces you've created before, and for quitting the application. Ibuild provides many such commands; we'll introduce them as we use them in this guide.

Directly above the pull-down menus are **edit tools** that help you edit interactor instances, either directly or in concert with pull-down menu commands. These tools are similar in concept to the creation tools, except they do not create new interactor instances. Rather, they affect *existing* instances in the viewing area. You can engage an editing tool just as you would a creation tool, that is, by clicking on its entry in the palette. Once engaged, you can use the tool on an existing instance by clicking on that instance. The tool's behavior after that initial click varies from tool to tool; we'll explain exactly what happens as we use the tools in subsequent examples.

Finally, a row of **indicators** lies above the editing tools. These indicators display information about your editing session:

- The **border width** indicator shows the border thickness that border and frame interactors will have when they are created. You can change the current border thickness by choosing a new thickness from the "Border" menu.
- The **color** indicator displays the foreground and background colors that newly created interactors will have. The indicator contains a rectangular outline whose color reflects the foreground color, and the outline is filled with background color. These colors can be changed independently with the ''FgColor'' and ''BgColor'' menus.
- The **interface name** indicator displays the interface's name. This name is used to store the interface in a file for later retrieval and editing. The interface is unnamed initially; ibuild prompts you for a name when you save the interface. Ibuild also prompts you for a name when you want to open and edit an interface created earlier. The commands for saving and opening interfaces are in the ''File'' menu.
- The **font name** indicator shows the name of the font that interactors will have when they are created. Of course, not all interactors display text; the font applies only to those that do. You can change the current font by choosing a new one from the ''Font'' menu.
- The **current magnification** indicator displays the viewing area's magnification factor. You may change the current magnification by zooming the viewing area in or out with the panner.

Assembling Hello's Interface

Hello's interface will simply display a string of text with a drop shadow. We'll build it step-by-step, describing new features along the way.

1. First you'll create the message that will display ''Hello world.'' But before you can create a message instance in the viewing area, you must engage the Message tool. Do so now by clicking on the word ''Message'' in the palette of creation tools.

2. Next, create the Message instance by clicking in the viewing area. You'll see a short vertical bar, called an **insertion point**, that prompts you to enter the text of the Message: type ''Hello world.'' If you make a mistake, just backspace over it and continue typing the text.
3. Next, increase the font size by pulling down the font menu and selecting ''Helvetica 18''.
4. Now add the drop shadow to the message you've created. Pull down the ''Composition'' menu, and you'll see an entry marked ''ShadowFrame.'' After you've chosen that entry, you should see the drop shadow around the Message. To make the shadow more pronounced, choose a thicker border from the Border menu.

If you've followed these steps, your screen should look like Figure 2 shown below.

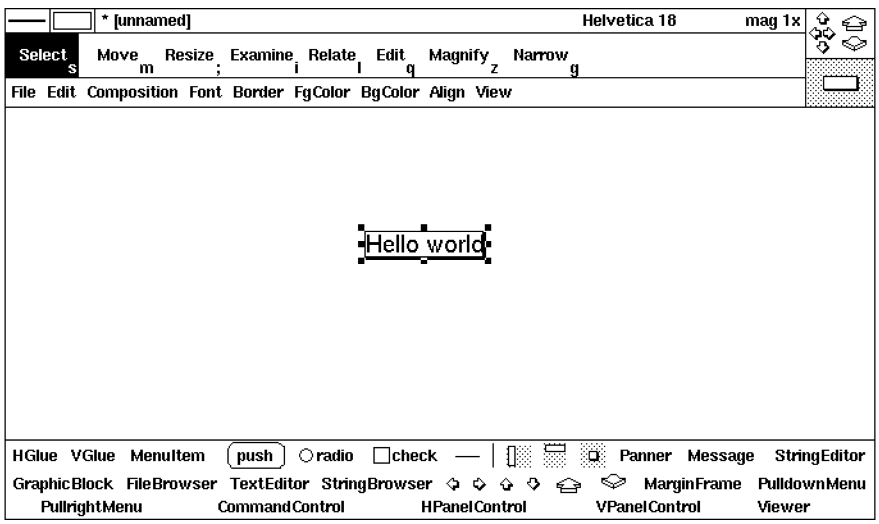

Figure 2.

Note that you haven't actually created the real interface, but you've told ibuild everything it needs to know to implement the real one. Now you can make ibuild generate the code for the real interface, compile it, and even run it for you. Then you can test it to make sure it behaves correctly.

In practice, however, the user interface is just a part of a larger application. Later we'll see how to integrate the code ibuild generates with the rest of an application.

Generating the Interface

Having assembled the interface, now you can tell ibuild to generate and compile the code that implements it.

1. First tell ibuild to generate the code for Hello by pulling down the File menu and choosing the ''Generate...'' entry.

2. A dialog box will appear (see Figure 3) that prompts you to name the interface, which we'll call ''Hello.'' Type that name into the dialog's text field and press the ''OK'' button.

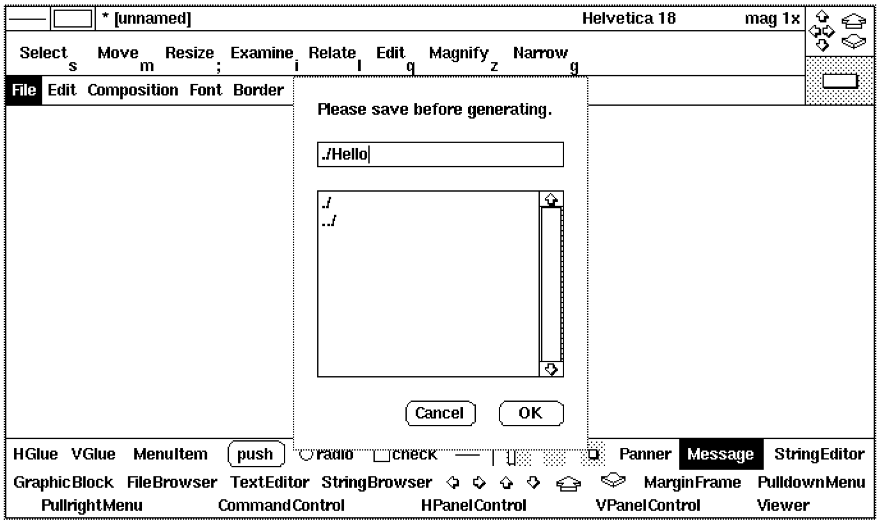

Figure 3.

3. Next, ibuild will display a dialog box that lets you specify the files to generate and whether or not to compile them into the real interface. By default, ibuild will generate all the necessary files and will then run the compiler, so you needn't do anything in this dialog box except press the ''OK'' button.

In response, ibuild will create several files that contain the information needed to compile a prototype of Hello. Then it will run the compiler to build an executable, which we can then run to test our interface.

When ibuild starts the compiler, it posts a dialog that reminds you that it is compiling the interface. The dialog also provides an ''Abort'' button that allows you to stop the compilation prematurely. This dialog disappears when compilation is complete. Then you can test the real interface.

Testing the Interface

To test our Hello prototype:

1. Pull down the File menu and choose the ''Execute...'' entry.
2. A dialog box will appear that prompts you to choose an executable to run from among those in the current directory. Choose ''Hello.exe'' and press ''OK.''

Ibuild will then execute the prototype interface, which you must position on the screen as you would any other interactive application in your windowing environment. As the program runs, you can test out the features of the interface, though the only thing you can do with our simple Hello interface is to resize it with the window manager. Ibuild posts a ‘‘Running Hello’’ dialog box to remind you that the prototype is running. When you have finished using the prototype, you can stop its execution by pressing the ‘‘Abort’’ button in this dialog box.

Summary

We’ve covered all the basic steps involved in creating an interface with ibuild except one: integrating the code it produced and the application code. We’ll cover that final step in a later chapter.

Here is a summary of the steps involved in designing and building an interface with ibuild:

- *Assembling the interface,* which involves choosing the interactors you’ll need and composing them into a complete interface that behaves correctly.
- *Generating the interface,* which creates the files needed to implement the final interface and optionally compiles them into a prototype executable.
- *Testing the interface*, which involves running the prototype executable and exercising the prototype to ensure that the code is correct. Note that this step is not essential to the building process because ibuild should not generate incorrect code. Nevertheless, you might want to test the real interface anyway.
- *Integrating the interface*, the step we did not take in this chapter, involves incorporating the code that ibuild generated into the rest of your application.

Hello’s interface is purposely trivial. We’ll build more interesting interfaces in the subsequent chapters.

Chapter 3 Creating Composite Interfaces

In this chapter we’ll use ibuild to construct **Namer,** a dialog box that prompts for a name (Figure 4). Namer’s interface is more sophisticated than Hello’s in that it is composed of several interactors, some of which handle input events such as mouse clicks and keystrokes. In contrast, Hello’s interface consisted of just one, output-only interactor.

Namer’s most important interface component is the text field into which the user types the name. We’ll use a StringEditor instance to implement this field, and a Frame instance will furnish the surrounding black outline. The user clicks a PushButton instance or hits the ‘‘Return’’ key to enter the name after it is typed. We’ll use a Message instance to display the fixed text ‘‘Name:’’ to the string editor’s left. The remaining interactors will simply compose these elements into Namer’s finished interface.

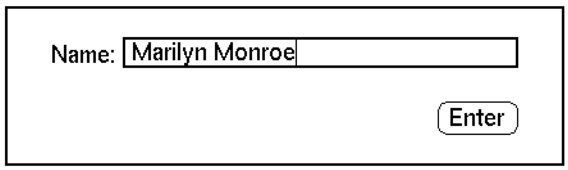

Figure 4.

Assembling Namer’s Interface

As with Hello, we’ll build Namer in several stages, explaining new features as we introduce them. If you make a mistake at any point, simply choose ‘‘Undo’’ from the Edit menu and continue.

1. If you still have Hello on your screen from the last chapter, you can start afresh by choosing ‘‘New’’ from the File menu.
2. We’ll create the fixed text ‘‘Name:’’ first. Click in the viewing area with the Message creation tool engaged, and then type the text.
3. Now create a StringEditor instance, which is similar to creating a message:
   a. Engage the StringEditor creation tool.
   b. Click in the viewing area.
   c. Type some sample text.

   You can be sloppy about where you place the string editor, because you’ll compose the objects with boxes later on. The sample text serves two purposes: it defines the string editor’s natural size, and it specifies a default string that will appear in the string editor initially. You may simply type spaces if there is no appropriate default.
4. Now put a frame around the StringEditor instance.
   a. First make sure the string editor is **selected**, that is, has eight small rectangles (called ‘‘handles’’) surrounding it. You may have noticed these handles in previous exercises, for example, after you created an interac-

tor with a tool. The handles identify the interactors that can be affected by menu operations. You can select any instance explicitly by clicking on it with the ‘‘Select’’ tool (in the editing tools palette) engaged.

b. With the StringEditor instance selected, choose the ‘‘Frame’’ entry from the Composition menu.

At this point your interface should look something like Figure 5.

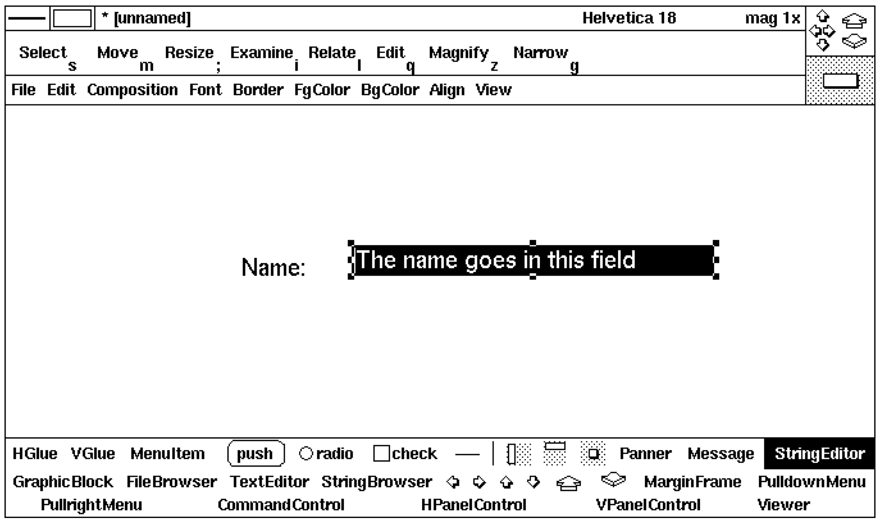

Figure 5. Message and StringEditor instances before HBox composition.

5. Next, use an HBox to tile the message and string editor horizontally.

a. Select both the message and the string editor. HBoxes compose selected objects left-to-right in the order they were selected. Select the Message first by clicking on it with the Select tool engaged. By default, selecting one object unselects all others currently selected. To select additional objects, hold the ‘‘Shift’’ key down as you click on them with the Select tool. Select the StringEditor instance in this manner so that both it and the Message instance have handles around them.

b. Compose the selections by choosing the ‘‘HBox’’ entry in the Composition menu.

Now the interface should resemble Figure 6.

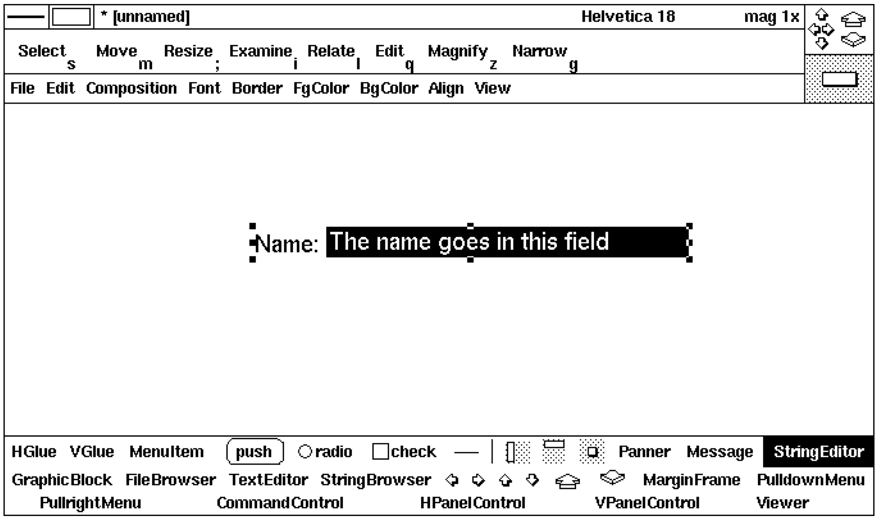

Figure 6. Message and StringEditor in an HBox.

6. Add a PushButton instance for entering the name.

   a. Click on the PushButton icon (‘‘push’’ with a rounded rectangle around it) in the creation tools palette to engage the PushButton tool.

   b. Click in the viewing area somewhere below the name field, and a PushButton instance will appear. Again, you needn’t position the button precisely since composition will determine its final position.

7. To right-justify the push button, put it in an HBox along with some HGlue to its left.

   a. Create the HGlue by clicking on ‘‘HGlue’’ in the creation tool palette and then clicking in the viewing area, holding down the left mouse button as you drag out a piece of HGlue of appropriate width.

   b. Release the mouse button when the width is satisfactory, and a piece of HGlue will appear. Ibuild adds horizontal stripes to make it easier to identify the glue as HGlue (VGlue has vertical stripes). Of course, real glue does not have stripes, and you can hide the striping of selected glue by choosing ‘‘Hide Glue’’ from the Edit menu.

   c. Compose the glue and push button in an HBox by selecting both of them (glue first) and choosing HBox from the Composition menu.

8. Change the string appearing in the PushButton instance from ‘‘push’’ to ‘‘Enter.’’

   a. Click on ‘‘Edit’’ in the editing tools palette to engage the Edit tool.

b. Click slightly to the right of the word ‘‘push’’ in the push button. An insertion point will appear indicating that you can edit the push button’s text: Simply erase the text with the ‘‘Delete’’ key and type the word ‘‘Enter.’’

9. Now compose the two HBoxes vertically in a VBox so that the HBox containing the message and string editor lies above the HBox containing the push button.

   a. A small piece of VGlue will provide some space between the two HBoxes. Create that piece now by engaging the VGlue creation tool and dragging out a piece about a centimeter tall.

   b. Compose the glue and HBoxes in a VBox by selecting them in top-to-bottom order: the HBox containing the message and string editor first, then the VGlue instance, and finally the HBox containing the push button. Remember to hold the shift key down when you click on the last two items to avoid unselecting the others.

   c. Choose ‘‘VBox’’ from the Composition menu.

10. Add a margin around the entire interface by placing it in a MarginFrame.

   a. Click on ‘‘MarginFrame’’ in the creation tools palette and then click anywhere on the interface as if to select it. Do not release the left mouse button.

   b. Now drag the mouse beyond the interface’s boundaries to vary the size of the margin.

   c. Release the mouse button when you’re satisfied with the margin’s size, and the interface will be surrounded by whitespace of that size.

11. Finally, put a border around the interface by selecting it and choosing Frame in the Composition menu.

The interface should now look like Figure 7. You can see how the interface will respond when it is resized by using the ‘‘Resize’’ tool in the editing tools palette. Simply engage the Resize tool and click on Namer’s interface, dragging the mouse as you would to resize it with a window manager. The interface will stretch and shrink to accommodate the size change. Note in particular how the vertical glue between the HBoxes stretches when you enlarge Namer—we’ll change this behavior later. You can revert the interface to its natural size by selecting it and then choosing ‘‘Natural Size’’ from the Edit menu.

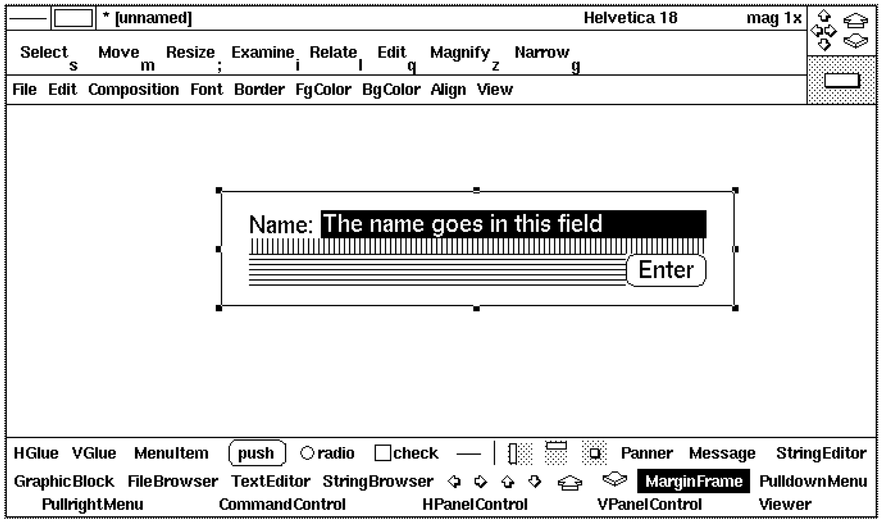

Figure 7. Completed Namer interactor composition.

Before moving on to the next section, save the interface you just created as ‘‘Namer’’ using the Save command in the File menu.

Examining and Editing Interactor Information

So far we have composed Namer’s interface, but we haven’t provided enough information to let Namer communicate with the rest of the application. For example, we probably want to give distinctive names to key interactor instances, such as the string editor, so that application code can access them. If you don’t specify such names, ibuild will make them up for you, and there’s no guarantee that ibuild’s names are mnemonic or even stay constant. We may also want to examine or change other attributes, such as an interactor’s shape. The ‘‘Examine’’ tool in the editing tool palette supports these tasks.

First, let’s focus on the active elements of Namer’s interface: the push button and the string editor. To examine the push button’s attributes:

1. Engage the Examine tool.
2. Click directly on the PushButton instance and hold the left mouse button down. You’ll see a pop-up menu containing two entries: ‘‘Info...’’ and ‘‘Props...’’
3. Move the mouse cursor over the Info... entry to reveal a submenu containing several additional entries: the bottom menu entry is the class name of the instance you clicked on; the entry above it is the parent instance’s class name; above that is the parent’s parent’s class name; and so on up to the root instance in the interface. Choose the ‘‘PushButton’’ entry, which should be bottommost, and release the mouse button.

A dialog box like that shown in Figure 8 will appear. It contains the information that characterizes the behavior of this particular PushButton instance. You cannot modify some of the information, such as the base class name, shape, and canvas, but you can give the instance a name and specify button state information. You can also supply the name of a member function to call whenever the button state is modified, which can communicate in turn with application code. We'll furnish all this information for the push button.

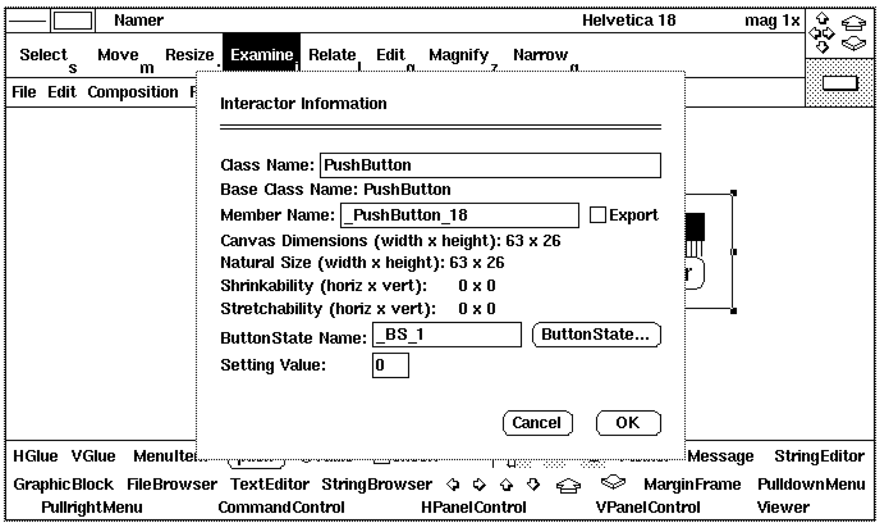

Figure 8. Initial PushButton attributes.

1. Name the push button by typing ‘‘enterButton’’ into the ‘‘Member Name:’’ field in the dialog box, and make sure there is an ‘‘x’’ in the ‘‘Export’’ check box. (Exporting this name makes this PushButton instance accessible to the application program; we'll see exactly how in the next chapter.)

2. Next, specify the name of a ButtonState instance to use. Do this by typing ‘‘enterBS’’ into the ‘‘ButtonState name’’ field in the dialog box. Note that the push button now only *refers* to a ButtonState instance named ‘‘enterBS’’; we haven't specified anything about that ButtonState instance yet.

3. Now let's supply information about the enterBS button state itself.

   a. Push the button labeled ‘‘ButtonState...’’ in the dialog box. Another dialog box titled ‘‘ButtonState Information’’ will appear (see Figure 9).

   b. Make sure the button state is exported; that is, the ‘‘Export’’ check box should have an ‘‘x’’ in it.

   c. Make ‘‘entered’’ the name of the member function to call whenever the button state is modified.

   d. At this point, your screen should look like Figure 9. Click ‘‘OK’’ in the ButtonState Information dialog.

4. Finally, specify the push button’s setting value by entering ‘‘1’’ in the Interactor Information dialog’s ‘‘Setting value’’ field, and press ‘‘OK.’’

.

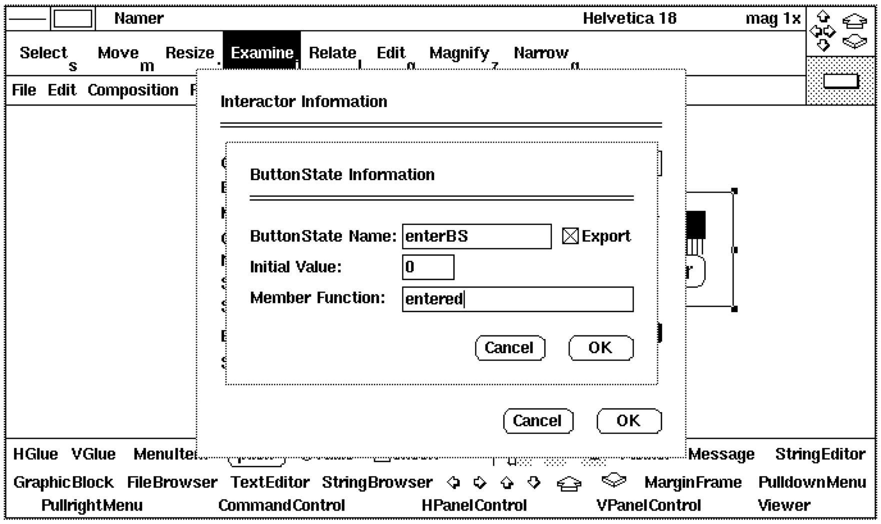

Figure 9. Modified ButtonState attributes.

Now let’s examine the string editor’s attributes and change its name.

1. Click directly on the StringEditor instance with the Examine tool engaged as you did before for the push button.
2. Move the mouse cursor over the Info... entry and choose the ‘‘StringEditor’’ submenu entry, which will be bottommost. A dialog box displaying the string editor’s attributes will appear.
3. Make ‘‘stringEditor’’ the string editor’s member name and make sure it is exported.
4. Press ‘‘OK.’’

Note that we didn’t specify any button state or member function information, and we still have to establish the equivalence between hitting the Return key and pressing the push button. Ibuild will set this information automatically when we relate the string editor to the push button with the Relate tool in the next section.

Finally, let’s change the shape of the VGlue separating Namer’s two HBoxes. Recall how the VGlue stretched as you resized the interface to make it larger. Now we’ll change the VGlue’s vertical stretchability so that it will not stretch; that is, it will be rigid.

1. With the Examine tool engaged as before, click directly on the piece of VGlue separating the HBoxes. The VGlue’s vertical stripes should make it easy to spot.

2. Move the mouse cursor over the Info... entry and choose the ‘‘VGlue’’ submenu entry, which will be bottommost. A dialog box displaying the glue’s attributes will appear.
3. Change the VGlue’s vertical stretchability to 0, thereby making it rigid.
4. Press ‘‘OK.’’

If you resize the interface now, you’ll see that the VGlue no longer stretches vertically.

Relating Interactors

Finally, we’ll tell ibuild that pressing the Return key is equivalent to clicking on the Enter button. If you were writing InterViews code, you’d specify this by having the push button and the string editor share their ButtonState instance. In ibuild, all you need to realize is that the string editor and the push button are related in this manner. The Relate tool lets you specify this relationship by direct manipulation.

The Relate tool affects two interactor instances, the **source** and the **target**. First you identify the source by clicking on it, and then you identify the target by clicking on it. As with the Examine tool, the Relate tool pops up a menu in response to each downclick to let you select the instance you are relating.

Now we’ll use the Relate tool to make the push button and string editor share a ButtonState instance:

1. Engage the Relate tool.
2. Click directly on the PushButton instance and hold the left mouse button down. You’ll see a pop-up menu with several entries, the bottommost being ‘‘PushButton.’’ Choose that entry and release the mouse button. Ibuild will draw a line from the push button to the current mouse position to remind you to specify the target.
3. Move the mouse cursor atop the StringEditor instance, click the left mouse button, and hold it down. You’ll see another pop-up menu with several entries, the bottommost being ‘‘StringEditor.’’ Choose that entry and release the mouse button.

The push button and the relate tool are now related; that is, they share the same button state. You can examine the string editor to verify that it uses the enterBS ButtonState instance (see Figure 10).

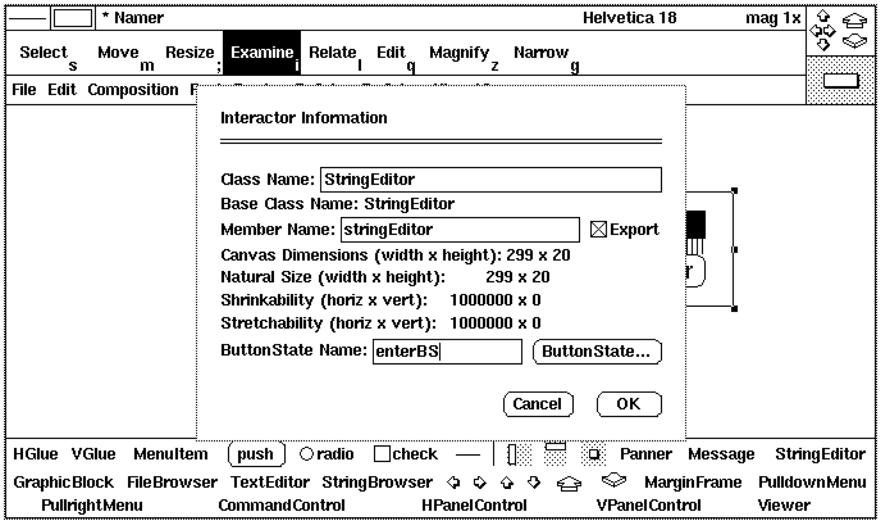

Figure 10. StringEditor attributes after relation to the PushButton instance.

Note that the push button was the source and the string editor was the target. This ordering is important because the target (the string editor) receives the ButtonState instance from the source (the push button), for which we specified information. Had we reversed the relation, the push button would have received the string editor's ButtonState instance, which we didn't specify.

The relate tool establishes relationships between other interactors as well. The semantics of the relationship depends on the two interactors and sometimes on the order in which the relation is made, but only one such relationship exists between any two interactors—there can be no ambiguity. For example, relating a scroller to a file browser makes the scroller scroll the file browser. Relating one radio button to another makes them share a ButtonState instance and also ensures that they mutually exclude one another. While we could specify these relationships solely through the Examine tool's dialog boxes, the Relate tool lets us establish the relationships quicker and more conveniently.

Chapter 4 Connecting the Interface to the Application

Ibuild lets you specify an application's user interface by direct manipulation instead of textual programming. Once you've specified the interface, ibuild can generate the code that implements it automatically. But the user interface is usually just a part of the application, which means that you must integrate the code that ibuild generates with the rest of the application's code.

This chapter describes ibuild's approach to integrating these two pieces of code. First we'll describe the integration mechanism in general. Then we'll describe the files that ibuild generates in more detail. Finally, we'll use the Namer interface from the previous chapter to demonstrate how to modify some of the files to support integration.

Core Classes and Subclasses

Ibuild can generate a C++ class called the **core class** for each top-level interface you've assembled in an ibuild session. (Our examples so far have had only one top level interface, but you can have any number of them.) By default, the core class is a subclass of MonoScene that defines the interactor composition. The main program (which ibuild generates for testing the interface) instantiates each class and inserts the instances into the world so that you can see and interact with them.

The core class contains all the exported member variables in the composition. For example, the exported button state ''enteredBS'' in Namer becomes a protected member of the core class. Moreover, all member function names specified in the interface (like the ''entered'' member function associated with enteredBS) have corresponding public virtual member functions in the core class.

Of course, protected members of the core class are not accessible to other objects, and the member functions do nothing by default. So how do these elements help us connect the user interface to the application? Because the core class is defined in ibuild-generated files, we might be tempted to modify these files directly to let the rest of the application access information in the core class. However, our changes will be lost the next time we edit the interface in ibuild and regenerate the core class files. We could rename the original files so that they are not overwritten, but then we'd be faced with the chore of re-editing the new files.

Ibuild addresses this problem by taking advantage of C++'s inheritance mechanism. In addition to the core class files, ibuild also generates files that define a **core subclass** for each core class. As the name suggests, each core subclass is derived from a core class. You modify the core *subclass*, not the core class, to establish an interface to the rest of the application. You can define member functions that provide controlled access to the exported member variables, and you can redefine the core class virtual functions to do useful work. If you later modify the interface's appearance with ibuild, you can choose to regenerate *only* the core class files, which you didn't modify: your changes to the core subclass are thus unaffected. Then you can recompile your application, and it will reflect the modifications you made to its interface. Only if you make radical changes to the interface (such as deleting interactors you've accessed in your core subclass code) should it become necessary to edit the core subclass code again.

Generated Files

When you name an interface, ibuild stores its representation in a file by the same name. For example, ibuild stored Namer in a file called ''Namer'' when we saved it. Generating the interface produces the C++ code that implements it. Ibuild distributes this code across several files, and the number of files depends on the number of core classes in the interface. Ibuild usually generates at least the following four files; each file name is prefixed by the name of the interface (e.g., Namer):

1. **Namer-main.c** defines a ''main'' routine for the prototype application, which lets you test the interface before you integrate it. This file is usually ignored following integration.
2. **Namer-props** defines the Xdefault properties for the interactor instances in the interface. Namer-main.c includes this file so that the prototype application's attributes reflect its appearance in ibuild. You will probably use this file to augment the properties of the final application.
3. **Namer-imake** is an Imakefile for building the prototype application. It may be discarded following integration, or you might use it as a starting point for the application's Imakefile.
4. **Namer-make** is the platform-dependent Makefile generated from the Imakefile, which actually builds the executable.

Specifying Classes

None of the files we've generated for Namer thus far contains core class or subclass declarations, because we didn't introduce any classes into the interface. We can do so by wrapping up Namer's top-level interactor (the frame) in a **MonoScene subclass instance**. Then we can examine that instance as we would any other, and we can change its class name to something mnemonic, like ''Namer.''

1. First select Namer's interface.
2. Choose ''MonoScene Subclass'' from the Composition menu. Namer's interface will now be wrapped in a MonoScene subclass, which will become the core subclass.
3. Click on the interface with the Examine tool and choose the Info... entry. The topmost entry in the resulting submenu should be an ibuild-generated class name such as ''_IB_MonoScene_0''; choose that entry.
4. Change the ''Class Name'' field to ''Namer.'' At this point, your screen should look like Figure 11. Press ''OK'' to continue.

Now save Namer (choose Save in the file menu) and generate the code (choose Generate... in the file menu). You needn't compile the generated code, since we'll modify it in the next section; so chose ''Generate source only'' in the ''Interface generation option'' dialog. Ibuild will generate four additional files, each prefixed by the MonoScene subclass name (''Namer'') we specified. These files declare and implement the core class and subclass:

1. **Namer-core.h** contains declarations for the core class. You should never modify this file, though you may need to look at it to see what members variables are exported.
2. **Namer-core.c** implements the core class, defining its interactor composition, member functions, etc. You should never modify this file; in fact, you should never need to look at it.

3. **Namer.h** declares the core subclass. You may modify this file to redefine or add new member functions or add new member variables.
4. **Namer.c** contains stubs for redefining all the virtual member functions defined in the core class. You may edit this file to implement the code that connects the interface to the application.

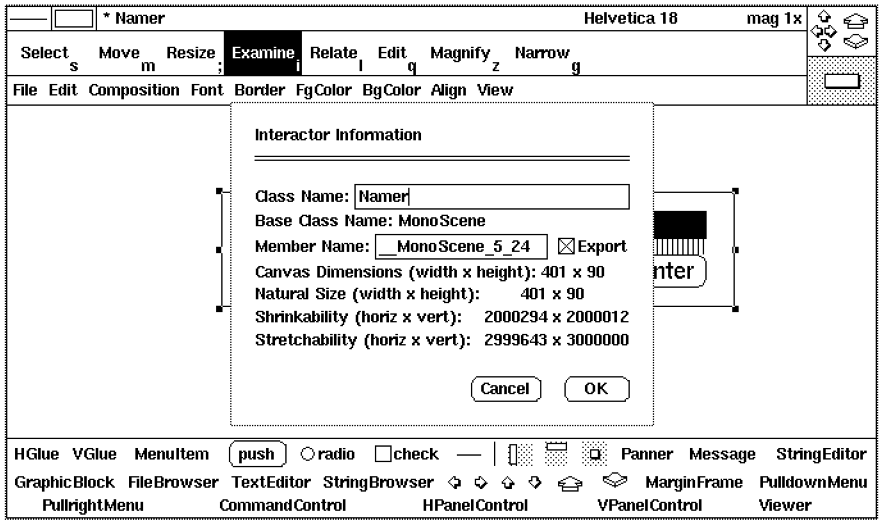

Figure 11. MonoScene subclass information.

If an interface contains more than one MonoScene subclass, ibuild will generate a corresponding set of four files for each MonoScene subclass, with each file name prefixed by the subclass name.

Note that you may change any interactor's class name to produce a core class and subclass. For example, you might change a push button's class name to ''MyPushButton.'' Doing so will produce a core class derived from PushButton and a corresponding core subclass named ''MyPushButton.'' Declarations and definitions for these classes will reside in **MyPushButton-core.h, MyPushButton-core.c**, **MyPushButton.h**, and **MyPushButton.c,** respectively. The push button that appears in the interface is actually an instance of MyPushButton, so you can specialize the button's behavior programmatically by modifying the core subclass.

Modifying the Core Subclass

Now we'll change the Namer core subclass to do some trivial work when the Enter button is pressed. While these changes are minor, they should give you a clue as to how a more sophisticated interface would communicate with its application.

Figure 12 shows the contents of Namer's core class header file. Notice the ''entered'' virtual member function we specified and the protected ButtonState, StringEditor, and PushButton instances we exported. The core subclass declaration in Namer.h (see Figure 13) simply derives from the core class and overrides the ''entered'' member function. We'll modify the core subclass implementation to print out the string editor's text when either the Enter push button or the Return key is pressed.

```
#ifndef Namer_core_h
#define Namer_core_h

#include <InterViews/scene.h>

class StringEditor;
class ButtonState;
class PushButton;

class Namer_core : public MonoScene {
public:
    Namer_core(const char*);
    virtual void entered() = 0;
protected:
    Interactor* Interior();
protected:
    ButtonState* enterBS;
    StringEditor* stringEditor;
    PushButton* enterButton;
};

#endif
```

Figure 12. Namer-core.h .

```
#ifndef Namer_h
#define Namer_h

#include "Namer-core.h"

class Namer : public Namer_core {
public:
    Namer(const char*);

    virtual void entered() = 0;
};

#endif
```

Figure 13. Namer.h

```
#include <InterViews/streditor.h>
#include <InterViews/button.h>
#include "Namer.h"
#include <InterViews/2.6/_enter.h>
#include <stream.h>

Namer::Namer(const char* name) : Namer_core(name) {}

void Namer::entered() {
    int value;
    enterBS->GetValue(value);
    if (value != 0) {
        cout << stringEditor->Text() << "\n";
        cout.flush();
        enterBS->SetValue(0);
    }
}
```

Figure 14. Modified Namer.c

Figure 14 shows the core subclass implementation file Namer.c with our modifications (shaded areas) that print out the string editor's text. We simply changed the ''entered'' member function (which will be called whenever enterBS is modified) to send the string editor's string to standard output. We also reset enterBS's value to 0 so that the push button will un-highlight itself, because push buttons highlight themselves when their button state's value equals the setting value.

With the changes in place, we can now use the Generate... command in the File menu to generate a prototype of the modified interface. Note that because the generated files already exist, ibuild will ask you if you want to overwrite them. Ibuild will overwrite all but the core subclass files by default, which is what we want both in this case and in general. You can also go back and change Namer's appearance in ibuild and regenerate the prototype without affecting our changes to Namer.c.

Summary

Ibuild leverages the C++ inheritance mechanism to decouple the ibuild-dependent portions of the interface code from the code that you modify to do useful work. The core class defines the interface's look and feel, and you do not modify this class. The core subclass lets you access and enhance the functionality in the core class without changing the core class interface or implementation. Together, core classes and subclasses offer a simple solution to the problem of keeping the builder's representation compatible with your changes to the generated code.

Chapter 5 Advanced Features

Now we'll discuss two of ibuild's advanced features: multiple views and user-defined creation tools. Both features are designed to streamline the interface building and editing process.

Multiple Views

Sometimes it's convenient to view an interface from two or more different perspectives at once. Perhaps you'd like to view the overall interface at a low magnification as you work on a small portion of it at high magnification. You might also want to change part of an interface's composition without disturbing unrelated parts. Ibuild's multi-view editing feature makes it possible to do both.

To view an interface in its entirety in another window, simply pull down the ‘‘View’’ menu and choose ‘‘New View.’’ Ibuild will produce a clone of the current window, and the clone will contain another view of the same interface. You can create any number of separate views in this manner. You can independently scroll and zoom the viewing area of each ibuild window. Changes to the interface appear synchronously in all its views. You can close an unneeded view by choosing ‘‘Close View’’ in its View menu.

Now let's return to our Namer interface. Suppose we want to let the user specify whether the name refers to a male or a female. Two radio buttons beneath the string editor would do the trick, but how do we put these radio buttons into an interface that's already composed?

We could start from scratch, or we could dissolve the scenes down to the level of the VBox and then add an HBox containing the radio buttons. But both of these alternatives force us to repeat work we've already done, which is undesirable even for small interfaces, let alone larger ones.

Instead, we can narrow our view of the interface to edit a portion of it out of context, as if it were the entire interface. We'll create a view of just the VBox and add the radio button composition to it.

1. Engage the ‘‘Narrow’’ tool in the editing tool palette and click on any component of the VBox, say the Enter push button.
2. A pop-up menu with ‘‘HBox’’ as the bottom entry (corresponding to the class name of the push button's parent instance) will appear. Choose the second entry from the bottom (‘‘VBox’’). Ibuild will narrow the view to display just the VBox instance, with its components individually selectable as if they hadn't yet been composed.
3. Now make some vertical room for the radio buttons by moving the HBox containing the Enter push button down a bit.
   a. Click on ‘‘Move’’ in the editing tools palette to engage the Move tool.
   b. Click on the HBox containing the Enter push button and drag it down a few centimeters. As before, the actual position of the interactors isn't important because the composition defines their placement, but moving them in this way gives us space to work more naturally. You can check the final composition at any time by selecting it and choosing Natural Size from the Edit menu.
4. Create the HBox containing the radio buttons.

a. Click on the RadioButton icon (‘‘radio’’ with a small circle to the left) in the creation tools palette to engage the RadioButton tool.

b. Create two RadioButton instances by clicking in the subview’s viewing area. Place them between the VGlue and the HBox containing the Enter push button.

c. Drag out a piece of HGlue to left-justify the radio buttons in the HBox.

d. Select the radio buttons and then the HGlue, and choose HBox from the Composition menu.

e. Use the Edit tool to change the radio buttons’ text to ‘‘Male’’ and ‘‘Female,’’ respectively.

At this point your screen should look like Figure 15.

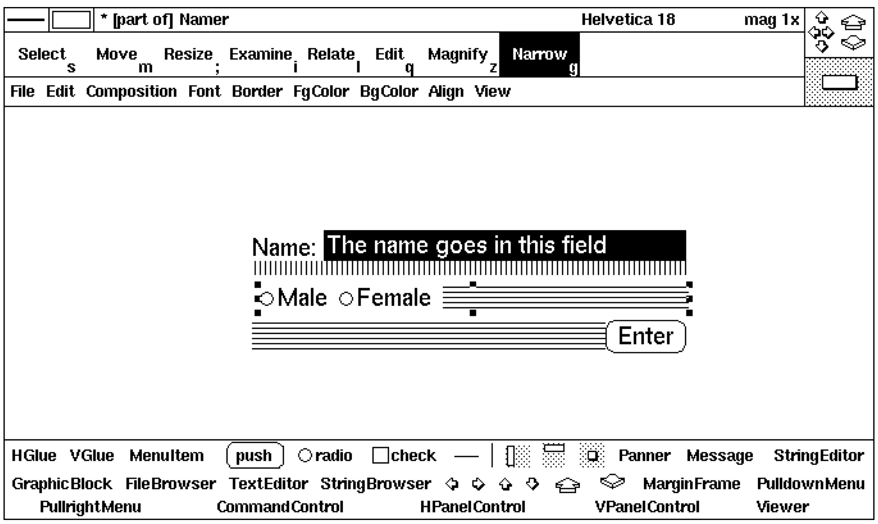

Figure 15. Adding radio buttons in a subview.

5. Reorder the elements of the VBox so that the radio buttons appear below the VGlue. Recall that VBoxes tile their components top-to-bottom. The HBox we just added is actually the *last* element in the VBox, since we added it last. Thus the radio buttons will appear *below* the HBox containing the Enter push button as soon as the VBox is given its natural size. We must therefore reorder the radio button HBox relative to the VGlue:

   a. Select the VGlue and the radio button HBox (VGlue first).

   b. Choose ‘‘Reorder’’ from the Composition menu. This command changes the relative order of selected instances to match the order in which they were selected.

   The interface should now appear at its natural size with the radio buttons falling between the VGlue and the HBox containing the Enter push button.

6. Examine the left most radio button (‘‘Male’’) to name its button state ‘‘genderBS’’ and its member function ‘‘gender.’’ You needn’t modify the setting value—keeping the initial and setting values the same will make this radio button the default.
7. Relate the two radio buttons so that they share a button state and mutually exclude one another. Make sure the ‘‘Male’’ radio button is the source and ‘‘Female’’ is the target.
8. Finally, return to the top-level view by choosing ‘‘View Root’’ from the View menu.

We have finished editing Namer’s appearance. Now save the interface and regenerate the code, overwriting all the files *except* Namer.c. If you look at the new Namer-core.h, you’ll see the ‘‘gender’’ virtual member function and the exported ‘‘genderBS’’ button state.

Now we’ll edit Namer.c to take advantage of the new functionality. We’ll override the ‘‘gender’’ virtual to print out a message whenever either gender radio button is pressed. The code for Namer.c appears in Figure 16 with our latest modifications in gray. Note that we didn’t have to declare this member function in Namer.h because ibuild did it for us when it overwrote Namer.h. And it was reasonable in this case to overwrite Namer.h because we never modified it.

```
#include <InterViews/streditor.h>
#include <InterViews/button.h>
#include "Namer.h"
#include <InterViews/2.6/_enter.h>
#include <stream.h>

Namer::Namer(const char* name) : Namer_core(name) {}

void Namer::entered() {
    int value;
    enterBS->GetValue(value);
    if (value != 0) {
        cout << stringEditor->Text() << "\n";
        cout.flush();
        enterBS->SetValue(0);
    }
}

void Namer::gender() {
    int sex;
    genderBS->GetValue(sex);
    if (sex == 0) {
        cout << "Male";
    } else {
        cout << "Female";
    }
    cout << " selected\n";
    cout.flush();
}
```

Figure 16. Namer.c modified to redefine gender member function.

Once you’ve made these changes, you can compile and test the prototype Namer interface. The prototype should look like Figure 17.

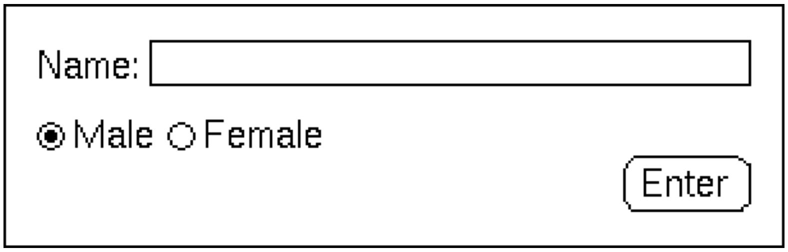

Figure 17. Namer with radio buttons added.

User-Defined Creation Tools

The creation tools palette initially contains a set of predefined tools. To augment this set, ibuild lets you define new creation tools for frequently-used interactor compositions. For example, suppose we need to use Namers in many different interfaces. We can introduce a creation tool that generates copies of Namer by choosing ‘‘Create Tool...’’ from the File menu. The dialog box shown in Figure 18 pops up to let you specify the appearance of the Namer tool’s icon in the creation tools palette.

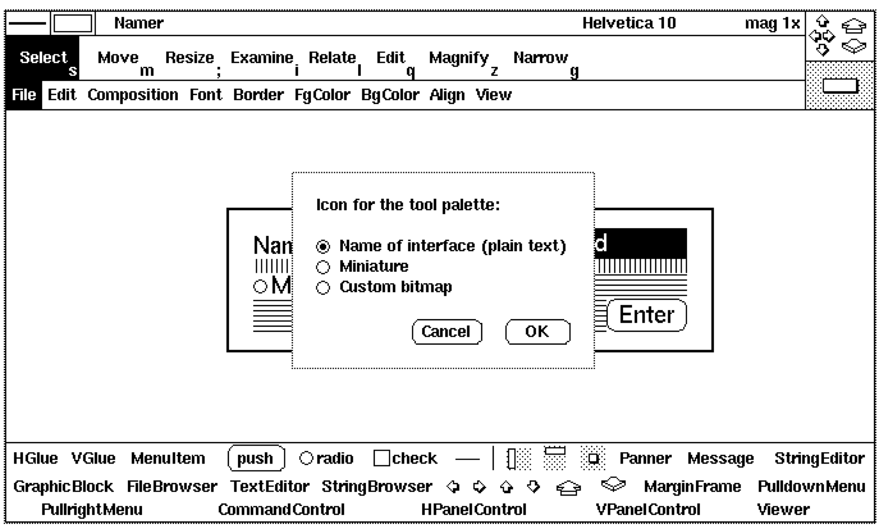

Figure 18. Specifying the icon for a user-defined creation tool.

If you click OK, the word ‘‘Namer’’ will appear in the creation tools palette. Alternatively, you could choose ‘‘Custom bitmap’’ to define a custom bitmap for the icon, or you could tell ibuild to display a miniature version of Namer itself in the icon by choosing ‘‘Miniature.’’

Ibuild also lets you remove unneeded tools from the creation tools palette and reinstall them should you need them later. To do this, choose ‘‘Tools...’’ from the File menu, and the **tool manager** dialog box shown in Figure 19 will appear.

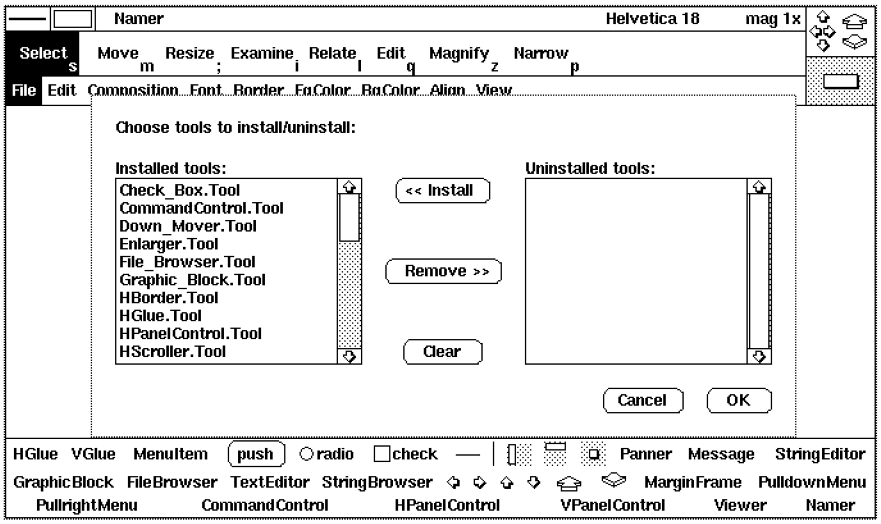

Figure 19. Tool manager dialog box.

To remove unneeded tools, simply click on their names in the left-hand browser and press the ‘‘Remove’’ push button. To install uninstalled tools, click on their names in the right-hand browser and press the ‘‘Install’’ push button. Press OK when you are done, and the creation tools palette will reflect your changes.

Chapter 6 Structured Graphics

Interactors such as scroll bars and menus have a fixed, predefined appearance; you cannot change the look of, say, a scroll bar in any significant way. However, ibuild provides an interactor subclass called **GraphicBlock** that displays any image you can create with idraw, InterViews's drawing editor. GraphicBlocks let you embellish your interface with custom graphics, such as a diagram or company logo. More interestingly, application code can access key graphical objects in a GraphicBlock. This lets you rearrange, replicate, animate, or otherwise manipulate the graphical objects programmatically, opening up a world of possibilities for graphical interfaces built with ibuild.

This chapter provides a simple but motivating example of what you can do with GraphicBlocks. In it we implement **hist** (Figure 20), an interactor that displays a **histogram**, or data in bar chart form. Hist's programming interface lets you add an unlimited number of labeled bars. Here we'll treat hist as a stand-alone interface, but it would more likely serve as a specialized interactor in a larger application.

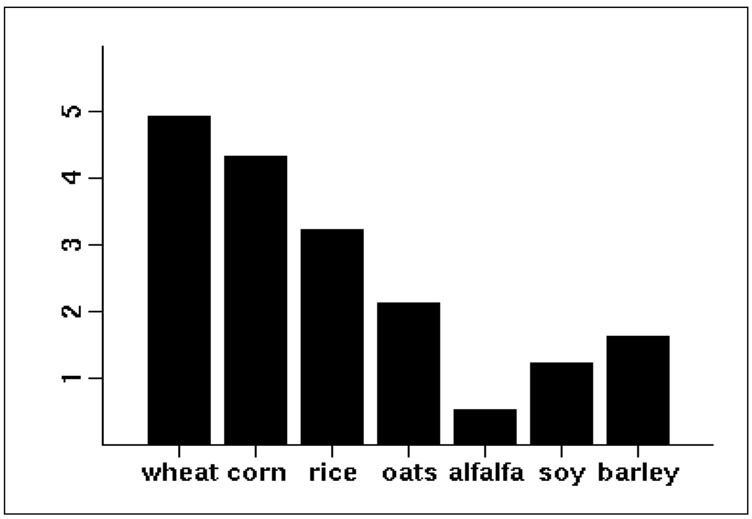

Figure 20. Histogram interface.

Hist's appearance can vary depending on the quality and quantity of data it displays. Because the interface isn't static, we obviously can't simply draw it and be done. But we will draw as much of the interface as we can interactively and then render its remaining visual attributes programmatically.

First we'll draw the fundamental elements of the histogram, that is, those parts that hist will always display regardless of the amount or kind of data. Figure 21 shows the basic histogram **cell**, which we'll draw in idraw and annotate in ibuild. Then we'll write code that replicates key parts of the cell to reflect the information we want to display in histogram form.

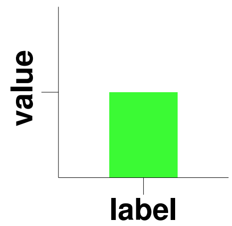

Figure 21. Basic histogram cell.

Creating and Examining a GraphicBlock

Before we can draw graphics in idraw, we must create the GraphicBlock instance that will contain them. But first, make sure you're editing a new interface by choosing New from the File menu. Then you can create the GraphicBlock.

1. Click on the word ‘‘GraphicBlock’’ in the palette of creation tools to engage the GraphicBlock tool.
2. Click in the viewing area and drag out a rectangle of arbitrary size. This rectangle will specify the GraphicBlock's initial size. The precise size is unimportant because ultimately the GraphicBlock will be just big enough to display the graphics it contains.
3. Release the mouse button when the rectangle's size is satisfactory, and the GraphicBlock of that size will appear.

The GraphicBlock is initially empty except for one rectangle object of the same size and shape as the GraphicBlock's canvas. This rectangle effectively props open the GraphicBlock; without it, the GraphicBlock would collapse to zero natural size. We'll replace the rectangle with the graphical elements that make up the cell by drawing them in idraw.

1. Engage the Examine tool.
2. Click on the GraphicBlock instance and choose the ‘‘Graphics’’ entry in the resulting pop-up menu. Ibuild will then start idraw.

You cannot use ibuild while idraw is active. Ibuild posts a dialog box to remind you of this. You can also abort idraw by pressing the ‘‘Abort’’ push button in the dialog.

Drawing the Interface

Now we can use idraw to draw the graphics that the GraphicBlock will display. In our case, we'll simply replace the initial rectangle with the graphics that make up the cell. Note that the following procedure takes place in *idraw*, not ibuild.

1. Delete the existing rectangle by selecting it and choosing ‘‘Delete’’ from the ‘‘Edit’’ menu.

2. Engage the Line tool and draw two lines, one horizontal and one vertical. (Hold down the Shift key as you draw to create perfectly horizontal or vertical lines.) These lines form the cell's axes. Arrange them so that they make an ''L'' shape like the axes in Figure 21.
3. Draw two relatively short lines, one horizontal and one vertical. These will represent the tick marks that demarcate a single unit along the horizontal and vertical axes.
4. Position the tick marks so that they coincide with the centers of the two axes. You may might find the commands in idraw's ''Alignment'' menu helpful in positioning the ticks precisely.
5. Engage the Text tool and create two text objects, one displaying the string ''label'' and the other ''value.'' (The text itself is not significant apart from helping us refer later to the text objects unambiguously.)
6. The ''value'' text object should be rotated 90 degrees. Do this now by selecting it and choosing ''90 CounterCW'' from the Edit menu.
7. Position the text objects so that they label the tick marks appropriately. Here again the alignment commands may prove useful.
8. Engage the Rectangle tool and sweep out a filled rectangle one unit high (on the value axis) and roughly as wide as the ''label'' text object. The rectangle should have a solid pattern; if it doesn't, select it and choose a solid pattern from the ''Pattern'' menu.
9. Position the rectangle so that it is centered horizontally with respect to the tick mark on the horizontal axis. The rectangle should also rest on the horizontal axis. At this point your screen should look something like Figure 22.

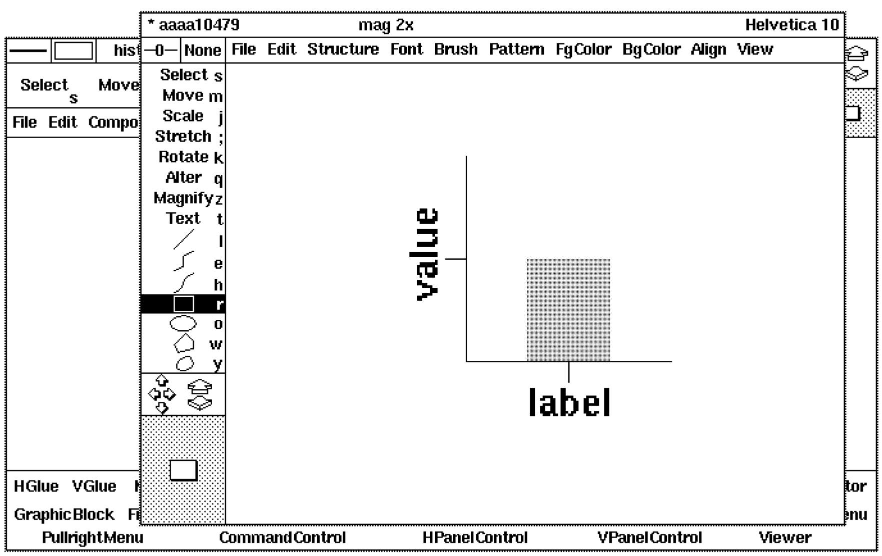

Figure 22. Drawing the histogram cell in idraw.

10. Finally, exit idraw by choosing ‘‘Quit’’ from the ‘‘File’’ menu. Idraw will ask you if you want to save the drawing before quitting. Press the ‘‘Yes’’ button to save your changes so that ibuild can incorporate them.

The cell you’ve drawn will now appear in the GraphicBlock as shown in Figure 23.

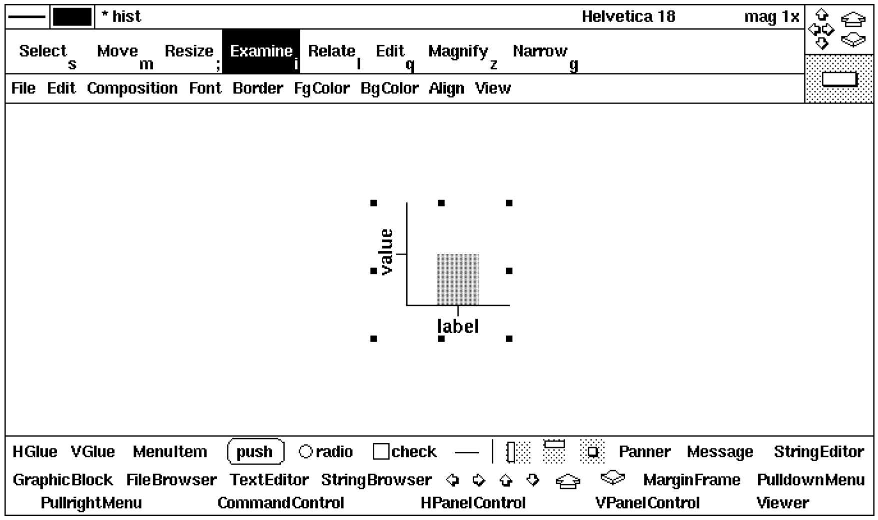

Figure 23. GraphicBlock displaying finished histogram cell.

Polishing the Interface

Before we leave ibuild, we must export several key objects to make them externally accessible. We must also introduce a core class so that we’ll have a core subclass to modify. Lastly, we’ll make ibuild generate the code.

1. First name the GraphicBlock and export it so that our code can access it.
   a. Examine the GraphicBlock.
   b. Type ‘‘_chart’’ into the member name field and check Export.
   c. Press ‘‘OK’’ to dismiss the dialog.
2. Compose the GraphicBlock in a ‘‘MonoScene subclass’’ as we have done before.
   a. Select the GraphicBlock and choose MonoScene Subclass from the Composition menu.
   b. Examine the MonoScene subclass just created.
   c. Type ‘‘Hist’’ into the class name field.
   d. Type ‘‘hist’’ into the member name field.
   e. Press OK.

3. Name and export the graphic objects in the cell. We can specify the names by examining each graphic and editing the resulting information dialog. For example, we would specify a member name of ‘‘_value’’ for the ‘‘value’’ text graphic and export it as follows:

   a. Engage the Examine tool and click directly on the ‘‘value’’ text graphic in the GraphicBlock.

   b. Type ‘‘_value’’ into the member name field and check Export.

   c. Press OK.

   Repeat this procedure for the other six graphical objects in the GraphicBlock. Name each element as shown in Figure 24.

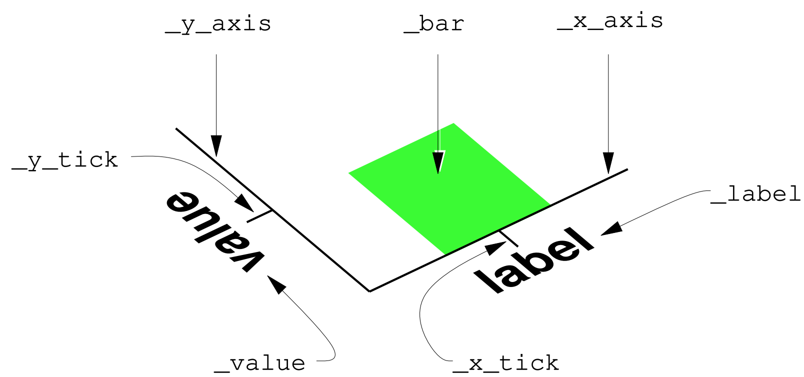

Figure 24. Exported member names of histogram cell components.

4. Generate the code for this interface by choosing the ‘‘Generate...’’ command from the File menu. Ibuild will ask you to save the interface before generating it. Save it under the name ‘‘hist.’’ By the way, there is no need to actually compile the interface at this point. You can bypass this step by selecting ‘‘Generate sources and makefile’’ in the ‘‘Interface generation options’’ dialog.

Programming the Interface

The cell contains only the most basic elements of the histogram. Now we must write code that replicates and modifies the basic elements to reflect the information in each bar of the histogram. More specifically, the code should

1. copy the `_y_axis` and `_y_tick` graphics and translate the copies to form a *y* axis of a given height,

2. label each tick mark on the *y* axis incrementally, and

3. provide a programming interface to adding a new bar to the histogram, extending the *x* axis as a side-effect.

Figure 25 shows the Hist core subclass declaration with our changes in gray. A programmer instantiates the Hist class by supplying the standard instance name string and an optional maximum *y* value. The Include member function adds a new bar to the histogram with the given value (as measured on the *y* axis) and textual label identifier, which will appear below the bar.

```
#ifndef Hist_h
#define Hist_h

#include "Hist-core.h"

class Hist : public Hist_core {
public:
    Hist(const char*, float max_value = 10.);
    virtual ~Hist();

    void Include(const char* label, float value);
private:
    void RemoveDummies();
    void InitValueAxis();
private:
    int _count;
    float _max_value;
};

#endif
```

Figure 25. Modified Hist.h core subclass declaration.

We've also added two private helper functions and two private member variables:

- The **RemoveDummies** function removes the basic cell elements from the GraphicBlock, since the actual elements will reflect the values specified in calls to the Include function. The original elements remain accessible, however, since they are members of the core class.
- The **InitValueAxis** function builds up the *y* axis when Hist is instantiated. The height of the *y* axis is fixed by the maximum value passed to the constructor. Therefore InitValueAxis is called only once.
- The **_count** member stores the number of bars that have been included (via Include).
- The **_max_value** member records the maximum *y* value supplied to the constructor.

Figure 26 presents the main routine for the program with our changes in gray. The constructor call is changed to specify a maximum *y* value of 5.0, and the histogram is given data to display through several calls to Include.

```
#include <Unidraw/catalog.h>
#include <Unidraw/unidraw.h>
#include <Unidraw/creator.h>
#include <InterViews/canvas.h>
#include <InterViews/painter.h>
#include <InterViews/sensor.h>
#include <InterViews/world.h>
```

```
#include <InterViews/perspective.h>
#include "Hist.h"
#include <InterViews/2.6/_enter.h>

static PropertyData properties[] = {
#include "hist-props"
    { nil }
};

static OptionDesc options[] = {
    { nil }
};

int main (int argc, char** argv) {
    Creator creator;
    Unidraw* unidraw = new Unidraw(
        new Catalog("/****/", &creator), argc, argv, options, properties
    );
    World* w = unidraw->GetWorld();
    Hist* hist = new Hist("hist", 5.0);

    hist->Include("wheat", 4.9);
    hist->Include("corn", 4.3);
    hist->Include("rice", 3.2);
    hist->Include("oats", 2.1);
    hist->Include("alfalfa", .5);
    hist->Include("soy", 1.2);
    hist->Include("barley", 1.6);

    w->InsertApplication(hist);
    unidraw->Run();
    delete unidraw;
    return 0;
}
```

Figure 26. Modified hist-main.c.

Figure 27 shows the modifications to the core subclass implementation. The constructor initializes the private members added to the core subclass and calls RemoveDummies and InitValueAxis to initialize the GraphicBlock's graphic contents. The destructor deletes all the graphics objects that RemoveDummies removed from the GraphicBlock. (The GraphicBlock will destroy the graphics that it contains.)

```
#include <Unidraw/iterator.h>
#include <Unidraw/Graphic/grblock.h>
#include <Unidraw/Graphic/lines.h>
#include <stdio.h>

#include <Unidraw/Graphic/polygons.h>
#include <Unidraw/Graphic/picture.h>
#include <Unidraw/Components/text.h>
#include "Hist.h"
#include <InterViews/2.6/_enter.h>

```

```
Hist::Hist(const char* name, float max_value) : Hist_core(name) {
    _count = 0;
    _max_value = max_value;
    RemoveDummies();
    InitValueAxis();
}

Hist::~Hist () {
    delete _x_axis;
    delete _y_axis;
    delete _x_tick;
    delete _y_tick;
    delete _label;
    delete _value;
    delete _bar;
}

void Hist::RemoveDummies () {
    Iterator i;
    Graphic* g = _chart->GetGraphic();
    g->First(i);

    while (!g->Done(i)) {
        g->Remove(i);
    }
}

void Hist::InitValueAxis () {
    float x0, y0, x1, y1;
    _y_axis->GetBounds(x0, y0, x1, y1);

    for (int i = 0; i < int(_max_value); ++i) {
        char value_buf[100];
        sprintf(value_buf, "%d", i+1);
        float dy = (y1 - y0)/2 * i;

        Graphic* y_axis = _y_axis->Copy();
        Graphic* y_tick = _y_tick->Copy();
        Graphic* value = new TextGraphic(value_buf, _value);

        _chart->GetGraphic()->Append(y_axis, y_tick, value);

        y_axis->Translate(0., dy);
        y_tick->Translate(0., dy);
        y_tick->Align(VertCenter, value, VertCenter);
    }
}

void Hist::Include (const char* label_string, float value) {
    Graphic* x_axis = _x_axis->Copy();
    Graphic* x_tick = _x_tick->Copy();
    Graphic* label = new TextGraphic(label_string, _label);
    Graphic* bar = _bar->Copy();

    _chart->GetGraphic()->Append(x_axis, x_tick, label, bar);

    float x0, y0, x1, y1;
    _x_axis->GetBounds(x0, y0, x1, y1);
```

```
        float dx = (x1 - x0)/2 * _count;

        x_axis->Translate(dx, 0.);
        x_tick->Translate(dx, 0.);
        x_tick->Align(HorizCenter, label, HorizCenter);

        bar->Scale(1., value);
        x_tick->Align(TopCenter, bar, BottomCenter);

        ++_count;
        _chart->SetGraphic(_chart->GetGraphic());
}

```

Figure 27. Modified Hist.c.

By definition, a GraphicBlock displays a graphic object. When ibuild creates a GraphicBlock instance, it passes a **Picture** instance to the GraphicBlock to display. A picture is a graphical object that has children. All the graphics that we add to the GraphicBlock actually become children of this picture. A programmer can retrieve the graphic in the GraphicBlock with its ‘‘GetGraphic’’ operation. RemoveDummies simply iterates over the graphics in the GraphicBlock’s picture, removing each in turn.

InitValueAxis builds up hist’s *y* axis by making copies of the `_y_axis` and `_y_tick` members and translating them to the proper position. The translation factor is a multiple of the height of the `_y_axis` graphic. InitValueAxis also creates a TextGraphic instance that displays the proper value for each tick on the axis, and it aligns the text with respect to the tick mark.

The Include function extends the *x* axis with each bar in a manner similar to what InitValueAxis does for the *y* axis. Include also copies the `_bar` object, scales it in the *y* dimension to reflect the bar’s value, and positions it properly by aligning it with the tick mark. Finally, Include increments the count of bars and updates the GraphicBlock to reflect the change in its picture, which presently can only be done by calling the GraphicBlock’s SetGraphic operation with its own graphic as the argument.

Chapter 7 Building a Complete Application

Our last example in this guide uses ibuild to create a simple but complete text editing application called **sted**. Figure 28 shows what sted looks like in use. Across the top of the sted window are three pull-down menus labeled ‘‘File,’’ ‘‘Edit,’’ and ‘‘Search.’’ The File menu lets you open and save a file, clear the current file, and quit the application. Edit contains the standard ‘‘Cut,’’ ‘‘Copy,’’ and ‘‘Paste’’ entries, while Search contains an entry for performing regular expression searches. Along the bottom and right are scroll bars that let you scroll the text shown in the center.

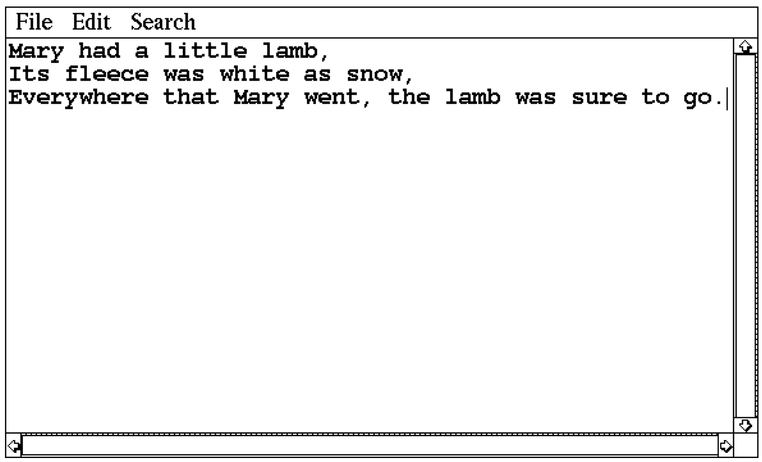

Figure 28. Sted editor.

Assembling Sted’s Interface

Sted’s interface requires four independent interactor compositions:

1. A top-level window made up of three PulldownMenu instances, two scrollers, four movers, and a TextEditor instance.
2. A dialog box that lets the user specify a file in which to save the text.
3. A dialog box that lets the user specify a file to edit.
4. A dialog box that lets the user specify a regular expression for searching the file.

At this point we’ll leap ahead and show you the ibuild screen with the finished compositions (Figure 29). With two exceptions, these compositions were built using the techniques already discussed, so we’ll omit most of the construction details.

The first exception involves the dialog boxes: Instead of wrapping them in a MonoScene subclass instance, we’ll use instead a **Dialog subclass instance** by choosing ‘‘Dialog Subclass’’ from the Composition menu. The semantics are the same as those for the MonoScene subclass instance, except ibuild will derive from Dialog instead of MonoScene. This lets us specify a button state for the dialog and redefine its Accept virtual function if necessary.

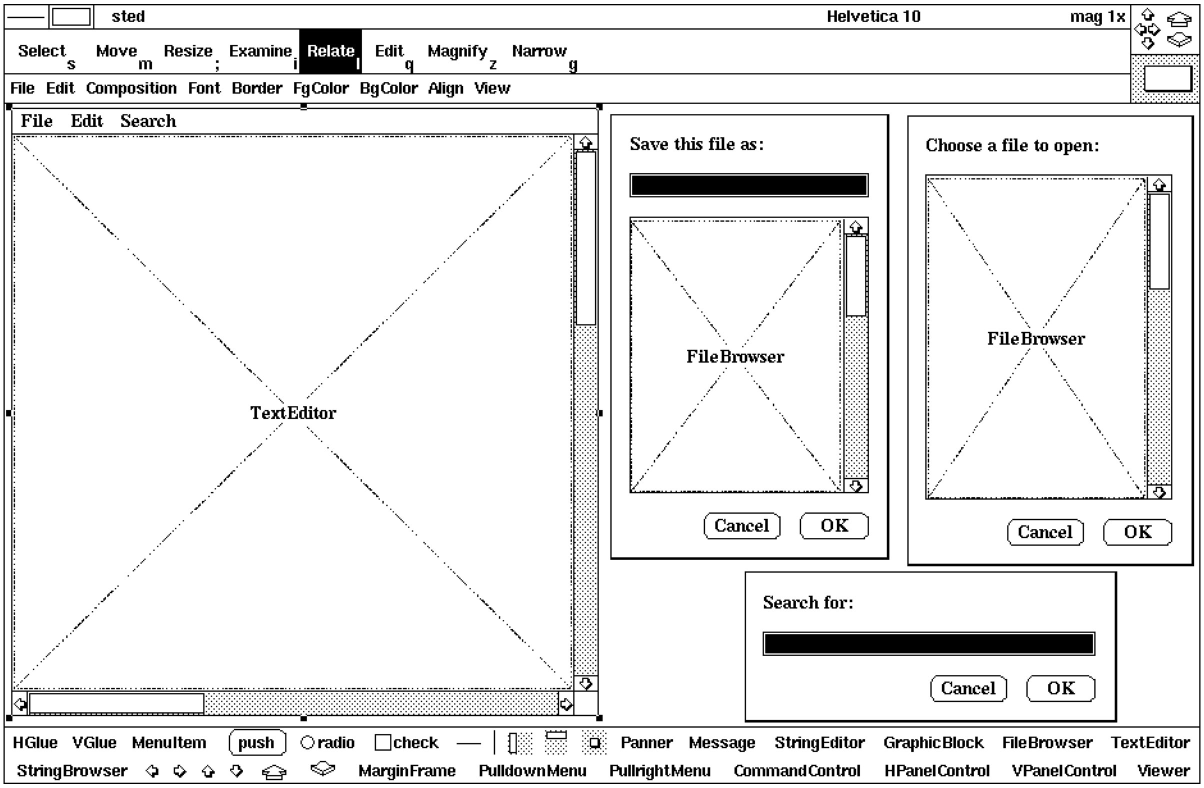

Figure 29. Sted interactor compositions.

The second new technique involves building pull-down menus. To create sted's menu bar of pull-down menus:

1. Engage the ''PulldownMenu'' tool in the creation tools palette and click in the viewing area to produce a PulldownMenu instance.
2. Use the Edit tool to change the pull-down menu's text from ''Pulldown-Menu'' to ''File.''
3. Engage the Narrow tool and click on the pull-down menu, holding the mouse button down. A pop-up menu will appear with ''PulldownMenu'' as the bottommost entry; move the cursor to that entry.
4. Release the mouse button, and the pull-down menu's text (''File'') will appear in reversed foreground and background colors. You are now editing the menu's body in a subview.
5. Add menu entries for clearing the editor (''New''), for opening a file for editing (''Open''), for saving the current file (''Save''), and for quitting the application (''Quit''). To create a menu entry:
   a. Engage the ''MenuItem'' tool in the creation tools palette.

b. Click in the viewing area somewhere below the pull-down menu ‘‘File’’ tag to create a MenuItem instance. As before, the exact position of each menu item will be determined by the composition, in this case, the pull-down menu.

c. Use the Edit tool to change the menu item’s text as appropriate.

At this point the File menu should look like Figure 30. Now return to the original (i.e., top level) view by selecting ‘‘View Root’’ from the View menu. The pull-down menu’s body is not visible normally; therefore the body disappears when you leave the subview.

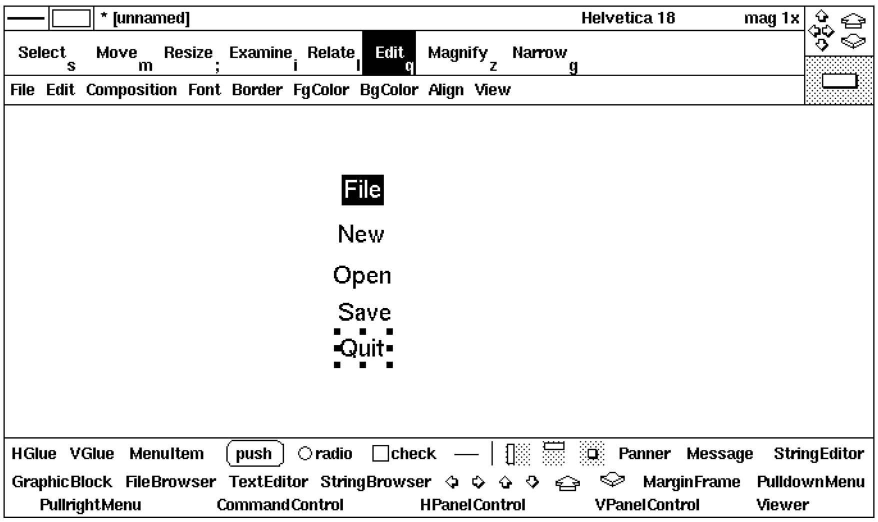

Figure 30. Composing a pull-down menu.

Create pull-down menus for the Edit and Search operations in the same manner. Once all three menus have been created, they can be composed in a menu bar:

1. Shift-select the three pull-down menus.
2. Choose the ‘‘MenuBar’’ entry from the Composition menu.

Now the three pull-down menus are composed in a MenuBar instance, which makes them work in concert.

Code Modifications

If you were to test sted’s interface as generated, all four compositions would appear on the screen at once. We’ll edit some of the generated files to make sted work correctly.

The first file to edit is the core subclass header file Sted.h. Figure 31 shows that file with our modifications in gray. The remaining core subclass modifications are shown in Figures 32 through 35.

```
#ifndef Sted_h
#define Sted_h

#include "Sted-core.h"

class Sted : public Sted_core {
public:
```

```
    Sted(const char*);

    virtual void New();
    virtual void Open();
    virtual void Save();
    virtual void Quit();
    virtual void Cut();
    virtual void Copy();
    virtual void Paste();
    virtual void Search();

    virtual void Handle(Event&);
private:
    void InsertChar(char);
    void InsertDialog(Interactor*);
    void RemoveDialog(Interactor*);
private:
    class FileManager* _mgr;
};

class FileManager {
public:
    FileManager();
    class TextBuffer* GetTextBuffer();

    void Open(const char* filename);
    void Save(const char* filename);

    void Copy(int beg, int end);
    const char* Clipboard();

    boolean FwdSearch(const char*, int& beg, int& end);
private:
    TextBuffer* _text;
    char* _buffer;
    int _size;
    char* _clipboard;
};

inline TextBuffer* FileManager::GetTextBuffer () { return _text; }
inline const char* FileManager::Clipboard () { return _clipboard; }

#endif
```

Figure 31. Modified Sted.h.Figure 32.

```
#ifndef OpenDialog_h
#define OpenDialog_h

#include "OpenDialog-core.h"

class OpenDialog : public OpenDialog_core {
public:
    OpenDialog(const char*);
    FileBrowser* GetFileBrowser() { return openFB; }
};
#endif
```

Figure 32. Modified version of OpenDialog.h.

```
#ifndef SaveDialog_h
#define SaveDialog_h

#include “SaveDialog-core.h”

class SaveDialog : public SaveDialog_core {
public:
    SaveDialog(const char*);
    StringEditor* GetStringEditor() { return saveSE; }
};

#endif
```

Figure 33. Modified version of SaveDialog.h.

```
#ifndef SearchDialog_h
#define SearchDialog_h

#include “SearchDialog-core.h”

class SearchDialog : public SearchDialog_core {
public:
    SearchDialog(const char*);
    StringEditor* GetStringEditor() { return searchSE; }
};

#endif
```

Figure 34. Modified version of SearchDialog.h.

```
#include <InterViews/texteditor.h>
#include <InterViews/textbuffer.h>
#include “Sted.h”
#include <InterViews/2.6/_enter.h>

#include <InterViews/event.h>
#include <InterViews/filebrowser.h>
#include <InterViews/regexp.h>
#include <InterViews/streditor.h>
#include <InterViews/textbuffer.h>
#include <InterViews/world.h>

#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>

#include “OpenDialog.h”
#include “SaveDialog.h”
#include “SearchDialog.h”

Sted::Sted(const char* name) : Sted_core(name) {
    _mgr = new FileManager;
    editor->Edit(_mgr->GetTextBuffer());
}
```

```

void Sted::Handle (Event& e) {
    if (e.eventType == KeyEvent) {
        if (e.len != 0) {
            char c = e.keystring[0];

            switch (c) {
            case '\010':
            case '\177':
                if (editor->Dot() != editor->Mark()) {
                    editor->DeleteSelection();
                } else {
                    editor->DeleteText(-1);
                }
                break;
            case '\015':
                InsertChar('\n');
                break;
            default:
                if (!iscntrl(c)) {
                    InsertChar(c);
                }
                break;
            }
        }
    } else if (e.eventType == DownEvent) {
        GetRelative(e.x, e.y, editor);
        editor->Select(editor->Locate(e.x, e.y));
        do {
            editor->ScrollToView(e.x, e.y);
            editor->SelectMore(editor->Locate(e.x, e.y));
            Poll(e);
            GetRelative(e.x, e.y, editor);
        } while (e.leftmouse);
    }
}

void Sted::InsertChar (char c) {
    editor->DeleteSelection();
    editor->InsertText(&c, 1);
    editor->ScrollToSelection();
}

void Sted::InsertDialog (Interactor* dialog) {
    World* world = GetWorld();

    Coord x, y;
    Align(Center, 0, 0, x, y);
    GetRelative(x, y, world);

    world->InsertTransient(dialog, this, x, y, Center);
}

void Sted::RemoveDialog (Interactor* dialog) {
    GetWorld()->Remove(dialog);
}

void Sted::New() {
    editor->SelectAll();
    editor->DeleteSelection();
}

void Sted::Open() {
    static OpenDialog* dialog;
```

```
    if (dialog == nil) dialog = new OpenDialog("openDialog");
    InsertDialog(dialog);

    FileBrowser* fb = dialog->GetFileBrowser();
    boolean accepted = dialog->Accept();

    RemoveDialog(dialog);

    if (accepted && fb->Selections() != 0) {
        _mgr->Open(fb->String(fb->Selection()));
        editor->Edit(_mgr->GetTextBuffer());
    }
}

void Sted::Save() {
    static SaveDialog* dialog;
    if (dialog == nil) dialog = new SaveDialog("saveDialog");
    InsertDialog(dialog);
    boolean accepted = dialog->Accept();
    RemoveDialog(dialog);

    if (accepted) {
        const char* name = dialog->GetStringEditor()->Text();
        _mgr->Save(name);
    }
}

void Sted::Quit() {
    exit(0);
}

void Sted::Cut() {
    Copy();
    editor->DeleteSelection();
}

void Sted::Copy() {
    _mgr->Copy(editor->Dot(), editor->Mark());
}

void Sted::Paste() {
    editor->DeleteSelection();
    editor->InsertText(_mgr->Clipboard(), strlen(_mgr->Clipboard()));
    editor->ScrollToSelection();
}

void Sted::Search() {
    static SearchDialog* dialog;
    if (dialog == nil) dialog = new SearchDialog("searchDialog");
    InsertDialog(dialog);
    boolean accept = dialog->Accept();
    RemoveDialog(dialog);

    if (accept) {
        int beg = editor->Mark(), end;

        if (_mgr->FwdSearch(dialog->GetStringEditor()->Text(), beg, end)) {
            editor->Select(beg, end);
        }

        editor->ScrollToSelection();
    }
}

```

```
static const int MINTEXTSIZE = 10000;

FileManager::FileManager () {
    _buffer = new char[MINTEXTSIZE];
    _text = new TextBuffer(_buffer, 0, MINTEXTSIZE);
    _clipboard = nil;
}

void FileManager::Open (const char* filename) {
    delete _buffer;
    delete _text;
    FILE* f = fopen(filename, “r”);

    if (f != nil) {
        struct stat filestats;
        stat(filename, &filestats);
        _size = max(round(filestats.st_size * 10), MINTEXTSIZE);
        _buffer = new char[_size];
        char* b = _buffer;
        int remaining = _size;

        while (remaining > 1 && fgets(b, remaining, f) != nil) {
            int l = strlen(b);
            remaining -= l;
            b += l;
        }
        fclose(f);
        _text = new TextBuffer(_buffer, b-_buffer, _size);

    } else {
        _size = MINTEXTSIZE;
        _buffer = new char[_size];
        _text = new TextBuffer(_buffer, 0, _size);
    }
}

void FileManager::Save (const char* name) {
    FILE* f = fopen(name, “w”);

    if (f != nil) {
        for (int i = 0; i < _text->Length(); ++i) {
            putc(_text->Char(i), f);
        }
    }

    fclose(f);
}

void FileManager::Copy (int beg, int end) {
    if (beg > end) {
        int tmp = beg;
        beg = end;
        end = tmp;
    }
    int sel = end - beg;

    if (sel != 0) {
        delete _clipboard;
        _clipboard = new char[sel+1];
        strncpy(_clipboard, _text->Text(beg, end), sel);
        _clipboard[sel] = ‘\0’;
    }
}

```

```
boolean FileManager::FwdSearch (const char* string, int& beg, int& end) {
    Regexp re(string);

    boolean found =
        _text->ForwardSearch(&re, beg) >= 0 ||
        _text->ForwardSearch(&re, _text->BeginningOfText()) >= 0;

    if (found) {
        beg = re.BeginningOfMatch();
        end = re.EndOfMatch();
    }
    return found;
}

boolean FileManager::BwdSearch (const char* string, int& beg, int& end) {
    Regexp re(string);

    boolean found =
        _text->BackwardSearch(&re, beg) >= 0 ||
        _text->BackwardSearch(&re, _text->EndOfText()) >= 0;

    if (found) {
        beg = re.BeginningOfMatch();
        end = re.EndOfMatch();
    }
    return found;
}
```

Figure 35. Modified Sted.c.

Finally, edit sted-main.c to keep instances of the various dialog boxes from appearing at once. Do this by simply deleting the corresponding lines of code. Figure 36 shows the lines *to be deleted* in gray.

```
#include <InterViews/painter.h>
#include <InterViews/shape.h>
#include <InterViews/sensor.h>
#include <InterViews/world.h>
#include “Sted.h”
#include “OpenDialog.h”
#include “SaveDialog.h”
#include “SearchDialog.h”
#include <InterViews/2.6/_enter.h>

static PropertyData properties[] = {
#include “sted-props”
    { nil }
};

static OptionDesc options[] = {
    { nil }
};

int main (int argc, char** argv) {
    World* w = new World(“****”, argc, argv, options, properties);
    Sted* __IB_MonoScene0_199 = new Sted(“__IB_MonoScene0_199”);
    w->InsertApplication(__IB_MonoScene0_199);
    OpenDialog* __IB_Dialog_200 = new OpenDialog(“__IB_Dialog1_200”);
    w->InsertASpplication(__IB_Dialog1_200);
    SaveDialog* __IB_dialog_201 = new SaveDialog(“__IB_Dialog1_201”);
    w->InsertApplication(__IB_Dialog1_201);
    SearchDialog* __IB_Dialog1_202 = new SearchDialog(“__IB_Dialog1_202”);
```

```
    w->InsertApplication(__IB_Dialog1_202);
    w->Run();
    delete w;
    return 0;
}
```

Figure 36. Lines to be deleted from sted-main.c.

You can now tell ibuild to compile the interface (being careful to avoid overwriting the files you changed), and sted becomes a fully functioning editor.

Adding New Features

We'll demonstrate how easy it is to modify the interface and extend sted's functionality by adding a backward search capability

1. First change the Search dialog's appearance in ibuild. Add two radio buttons to the Search dialog just as we did for Namer in the last chapter, except include a message labeled ''Direction:'' and label the radio buttons ''forward'' and ''backward.'' Use the Narrow tool to create a subview of the composition just as before. The new composition should look like Figure 37.

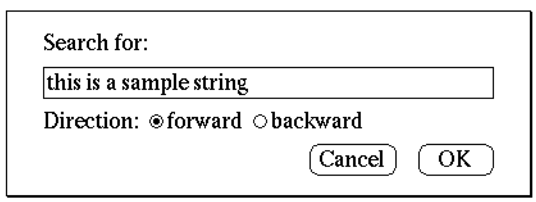

Figure 37. Search dialog box with radio buttons added.

2. Name the ''forward'' radio button's button state ''direction.'' You needn't name its member function, as we won't need to do anything special the moment the button state changes.
3. Relate ''forward'' to ''backward'' as you did for Namer's radio buttons.
4. Next, add a helper function to the Search dialog's core subclass that returns whether or not the search is in the forward direction, based on the value of the ''direction'' button state. Figures 38 and 39 present SearchDialog.c and SearchDialog.h with the required modifications in gray

```
#include <InterViews/streditor.h>
#include "SearchDialog.h"
#include <InterViews/2.6/_enter.h>

#include <InterViews/button.h>

SearchDialog::SearchDialog(const char* name) : SearchDialog_core(name) {}

boolean SearchDialog::ForwardSearch () {
    int value;
```

```
    direction->GetValue(value);
    return value == 0;
}
```

Figure 38. Final version of SearchDialog.c.

```
#ifndef SearchDialog_h
#define SearchDialog_h

#include "SearchDialog-core.h"

class SearchDialog : public SearchDialog_core {
public:
    SearchDialog(const char*);
    StringEditor* GetStringEditor() { return searchSE; }
    boolean ForwardSearch();
};

#endif
```

Figure 39. Final version of SearchDialog.h

```
class FileManager {
public:
    FileManager();
    class TextBuffer* GetTextBuffer();

    void Open(const char* filename);
    void Save(const char* filename);

    void Copy(int beg, int end);
    const char* Clipboard();

    boolean FwdSearch(const char*, int& beg, int& end);
    boolean BwdSearch(const char*, int& beg, int& end);
private:
    TextBuffer* _text;
    char* _buffer;
    int _size;
    char* _clipboard;
};
```

Figure 40. Sted.h modified for backward searching.

5. Now edit Sted.h to declare a helper function ‘‘BwdSearch.’’ The modifications appear in Figure 40
6. Edit Sted.c to implement the backward search (Figure 41).

```
void Sted::Search() {
    static SearchDialog* dialog;
    if (dialog == nil) dialog = new SearchDialog("searchDialog");
    InsertDialog(dialog);
    boolean accept = dialog->Accept();
    RemoveDialog(dialog);

    if (accept) {
        if (dialog->ForwardSearch()) {
```

```
            int beg = editor->Mark(), end;

            if (_mgr->FwdSearch(dialog->GetStringEditor()->Text(), beg, end)){
                editor->Select(beg, end);
            }
        } else {
            int beg = editor->Dot(), end;

            if (_mgr->BwdSearch(dialog->GetStringEditor()->Text(), beg, end)){
                editor->Select(beg, end);
            }
        }

        editor->ScrollToSelection();
    }
}
```

Figure 41. Sted.c modified for backward searching.

7. Finally, re-compile this new version to verify that the backward search works.

Appendix Shortcuts

Ibuild defines many keyboard shortcuts to facilitate repetitive operations. For example, a keyboard equivalent exists for most pull-down menu commands. Ibuild also provides shortcuts for other common operations as summarized in the tables below.

Viewing Area

General

Shortcut	Equivalent Operation
Control-Middle mouse	Grab-scroll viewing area.
Control-Right mouse	Rate-scroll viewing area.
Middle-mouse	Engage the Move tool temporarily.
Right-mouse	Engage the Select tool temporarily.
Shift-HGlue/VGlue	Create rigid glue.
Shift-MarginFrame	Create a rigid MarginFrame.
Shift-Move	Constrain movement to one dimension.
Shift-Narrow	Descend one level without displaying a pop-up menu.
Shift-Resize	Constrain resizing to one dimension.
Shift-Select	Extend selection.
TAB	Select the first (bottommost) interactor in the view if nothing is selected currently; otherwise select the interactor immediately after the current selection. An interactor containing text will become editable following this operation.

Text-Oriented Tools

Shortcuts for text-oriented tools such as the PushButton creation tool and the Edit tool:

Shortcut	Equivalent Operation
Control-A	Move the insertion point to the beginning of the text.
Control-B	Move the insertion point backward one character.
Control-D	Delete the text currently selected; if none, then delete the character following the insertion point.
Control-E	Move the insertion point to the end of the text.
Control-F	Move the insertion point forward one character.
Control-H or DEL	Delete the text currently selected; if none, then delete the character preceding the insertion point.
Control-K	Kill line.
Control-U	Select all.
Control-W	Select the preceding word.
ESC	Terminate text editing operation.

Dialog Boxes

General

Shortcut	Equivalent Operation
Control-G	Pushing “Cancel.”
Return	Pushing “OK.”
TAB	Switch keyboard focus.
n	Pushing “No” or “Don’t Generate.”
y	Pushing “Yes.”

Browsers

Shortcuts that apply to scrollable lists of choices, such as the list of files in the “Open...” command dialog:

Shortcut	Equivalent Operation
<	Select topmost-visible entry.
>	Select bottommost-visible entry.
DEL or BS	Unselect all.
G	Go to the last entry.
Middle-mouse	Grab-scroll.
Right-mouse	Rate-scroll.
SPACE	Scroll down one screenful.
a	Select all.
b	Scroll up one screenful.
d	Scroll down one-half screenful.
g	Go to the first entry.
j	Scroll down one entry.
k	Scroll up one entry.
n	Select next entry.
p	Select previous entry.
u	Scroll up one-half screenful.

Text Fields

Shortcuts that apply to text entry fields, such as the field for an interactor’s member name in its ‘‘Interactor Information’’ dialog:

Shortcut	Equivalent Operation
Control-A	Move the insertion point to the beginning of the text.
Control-B	Move the insertion point backward one character.
Control-D	Delete the text currently selected; if none, then delete the character following the insertion point.

Shortcut	Equivalent Operation
Control-E	Move the insertion point to the end of the text.
Control-F	Move the insertion point forward one character.
Control-H or DEL	Delete the text currently selected; if none, then delete the character preceding the insertion point.
Control-U	Select all.
Control-W	Select the preceding word.

Index