

## Appendix A. Build a Text Editor

This Appendix will walk you through building a simple application. Most of the illustrations will be screen shots from a prototype of the construction tool (itself a computer application), called the *Software Construction Set*.

Figure A.1 shows the two windows presented by the Software Construction Set. There is the large *construction window* where the program is built, and at the lower right is a little window called the *Control Panel*.

You will work only with the left-hand button of the Control Panel, the one shown with the red traffic light. Clicking this button stops and starts the program you are building, and the color of the traffic light alternates between red and green.

The vertical left-hand pane of the main window, the one with the vertically arranged icons, is the *Component Palette*. There are several of these palettes, selected by the list box at the lower left-hand corner of the large window. The element of the list box which has been highlighted in the figure is called “U.I.[user interface]-Windows”. You will be building the application by dragging icons from one or more palettes to the workspace (the big pane) and wiring them up.

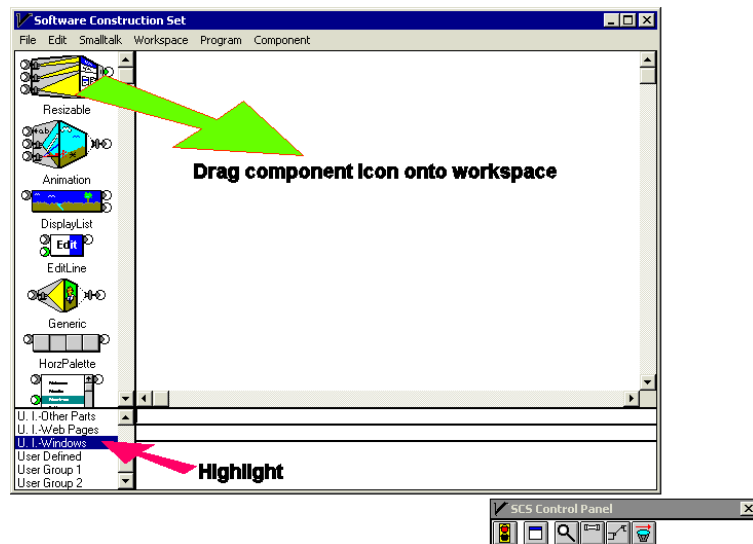




Figure A.1

The construction window and the control panel of the Software Construction Set

## The Two-component Starting Point

Figure A.2 below shows the starting point for most applications. The application you have built with these two components opens an empty window when you click the traffic-light button  on the control panel and turn it green.

The Resizable Window component has two *send connectors* (on the right side; they output flow objects) and three *receive connectors* (on the left side; they receive flow objects). The top send connector is responsible for opening the application window (i.e., the window that is projected by the Resizable Window component) and the bottom connector is responsible for closing the application window. Later we'll consider the "When Go" component  at the right end of the wire. For now it's sufficient to know that the When Go component is responsible for telling the Resizable Window component to open the window it is projecting *when you click the traffic-light button from red to green*.

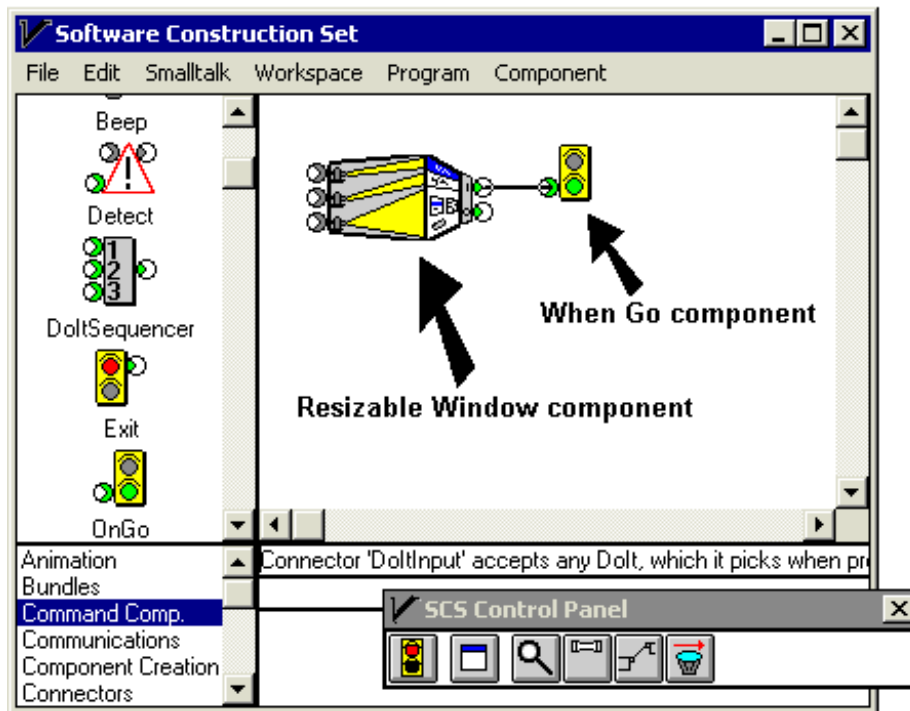


Figure A.2  
The starting point of most applications

## Run The Application

Figure A.3 shows the computer display after you click the traffic-light button. The traffic light has turned green, indicating that the application has been started. A new window appears, with “untitled” in its title bar. *This is the window projected by the two-component application you have built, which is now running.* (The windows have been resized and made to overlap in order to fit the picture to the page.) The function of the Resizable Window component at the left of the wiring diagram is to project the border of this application window.

Your task now is to add features to the application window, one by one.

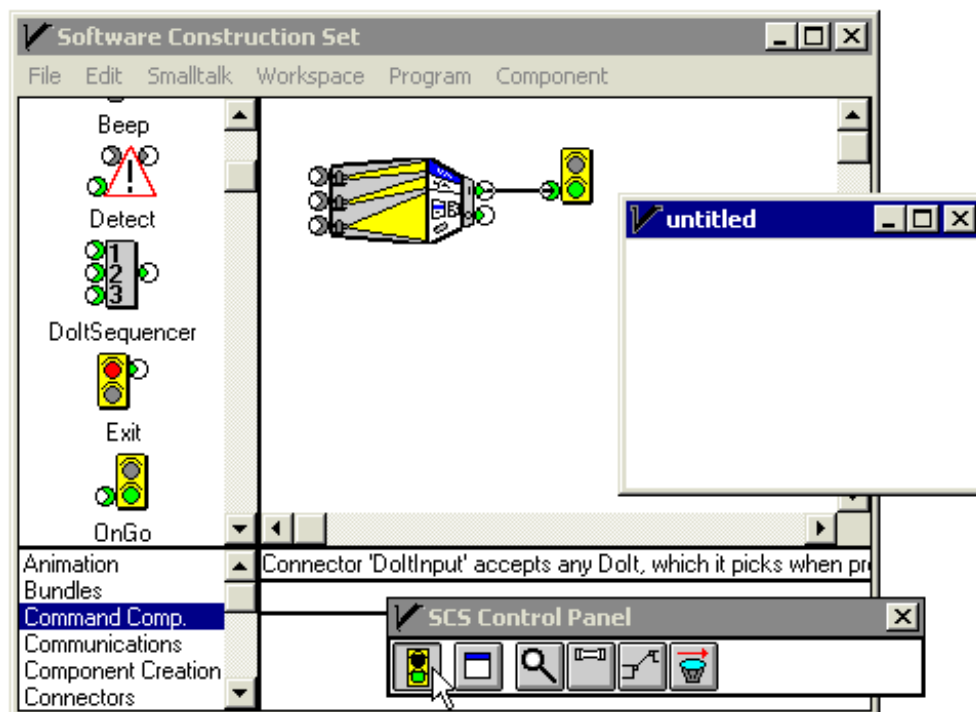
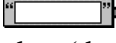


Figure A.3  
The window projected by the running program

## A Text Flow object

First you put some text into the title bar.

The “Text Source” component  sends a text flow object which is fed into the title bar (the top) receive connector of the Resizable Window component. You see the effect of this text flow object in the application window.

Now you see that, in this object-flow model of software, *a flow object is simply a carrier of data*. The flow object that flows from the Text Source component to the Resizable Window component carries data whose value is the character sequence “Fred”. The data carried by a flow object can be any object; in particular it can be a “business object” that has a message-based interface with those components that interact with it.

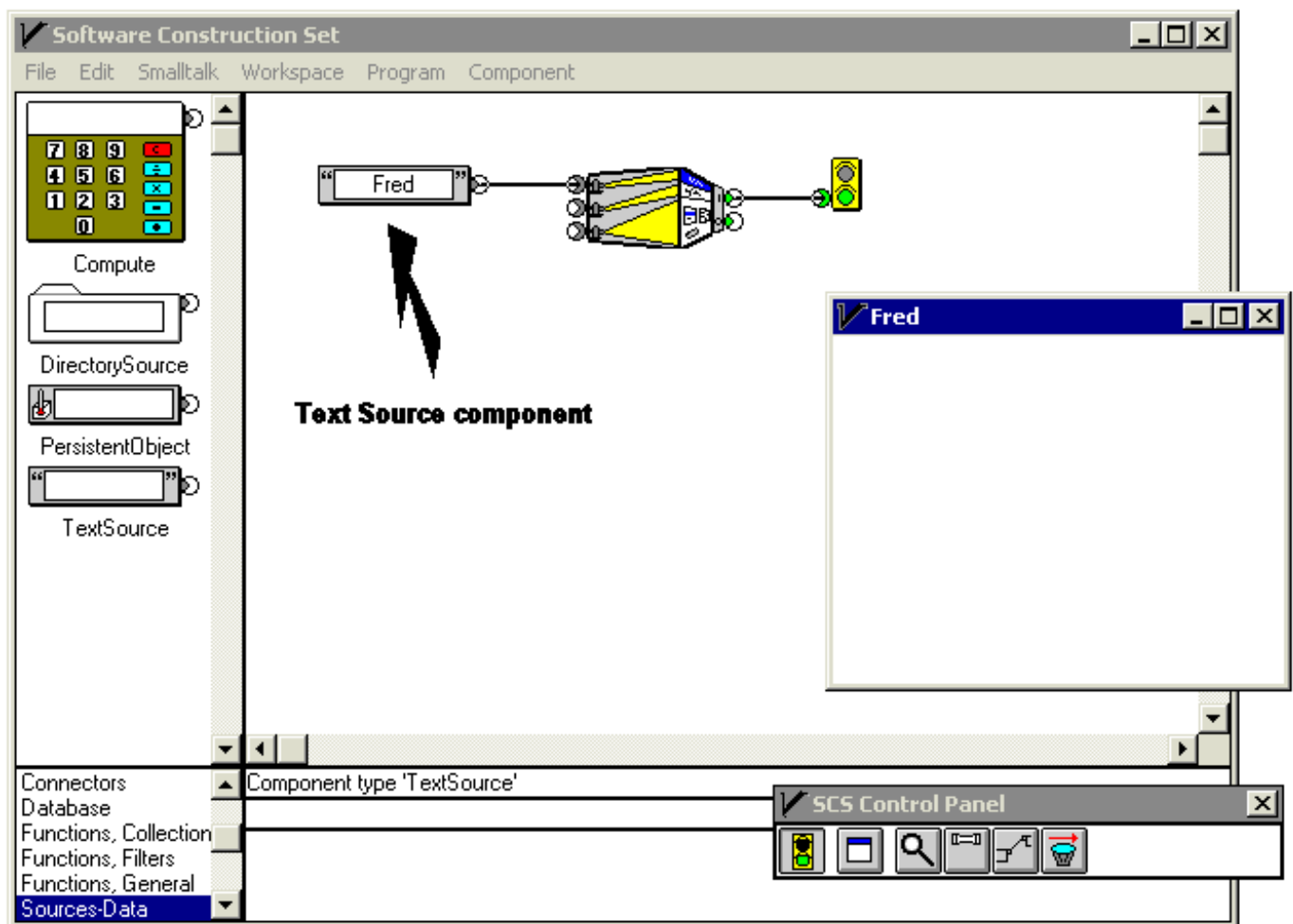


Figure A.4  
A title has been added to the window

## The Command Flow Object

Now that you know that the flow objects that flow along wires are simply carriers of data, we can address the flow object that runs from the Resizable Window component to the When Go component. This flow object is called a command flow object, because the kind of data it carries is called a *command*.




Figure A.5

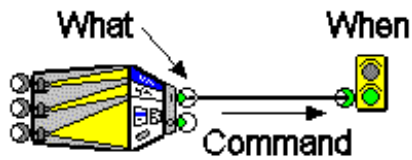
For every flow object there is one send connector in the total wiring diagram from which that flow object originates. For every command flow object, *its originating send connector is associated with a particular action of the component this send connector is attached to*. For example, in Figure A.5 the Resizable Window component's action associated with the top send connector is to open the window the component is projecting.

Each send connector of the Resizable Window component immediately puts out its respective command flow object. If there is a wire connected to that send connector the flow object will flow down that wire. The job of every command carried by a command flow object is to “stay in touch” with its originating send connector, and to tell that connector when to tell its component to execute the action with which the connector is associated, in this case, opening the application window. The command flow object, in its travels through the wiring which carries it, will be received by one or more components which are able to tell the command flow object's command when to tell its originating send connector to start the action.

Here is how it works in Figure A.5. The When Go component can, in effect, say “Now!” to any command flow object that flows into it. This is called “picking” the command flow object's command. When a command is picked, the command says “Now!” back to the send connector from which it originated. This originating send connector then causes its particular action to be performed by its component, in the case of Figure A.5, opening its application window.

What tells the When Go component *when* to pick the command? The particular function of the When Go component is to pick whatever command it receives when the person using the Software Construction Set clicks the traffic light button  in the Control Panel and changes it from red to green.

## A Discussion of Commands



A component that sends a command flow object has a send connector from which the command flow object originates; that send connector determines *what* the action is that the command will cause the component to perform. But another component, one which receives the command flow object, determines *when* that action will occur. In Figure A.5 and to the left, the *what* is determined by the top send connector of the Resizable Window component, and the *when* is determined by the When Go component.

When the receiving *when* component decides to pick a command it has received, the following sequence of events occurs.

1. The receiving *when* component picks the command carried by a command flow object the receiving component has received.
2. The picked command then tells its originating send connector to execute its action. (The command and the receiving component don't know what that action is, nor does the receiving component know which component emitted the command flow object.)
3. The originating send connector tells its component to perform the *what* action associated with that send connector.

The concept of a command flow object is unique to this conceptual model. It may look a little strange to those who are used to thinking that control information flows in the direction that the above sequence of events occurs, i.e., from right to left, at the time of the pick. In our way of thinking, the command flow object flows in the same way that all flow objects do, from left to right, and the flow is complete before the pick. The above sequence of events is not treated as a flow but simply as the behavior common to all commands. This treatment of a command like any other piece of data is important to the power of the conceptual model.

The sending component of a command flow object determines *what* but not *when*, the picking component determines *when* but not *what*, and the wiring makes the connection between the two by carrying a command from the first to the second. In principle, it is possible to wire any *what* component to any *when* component, and the *what* component will execute its action when the *when* component picks the command it has received. *Neither component knows or cares about the identity of the other.* This is called *decoupling* of the *what* and *when* functions of the application. Decoupling is good because it simplifies reuse of components.

## Add Menus

Now that commands are out in the open you can add menus to the application window. You are only going to add the File/Open menu item and a few Edit menu items.

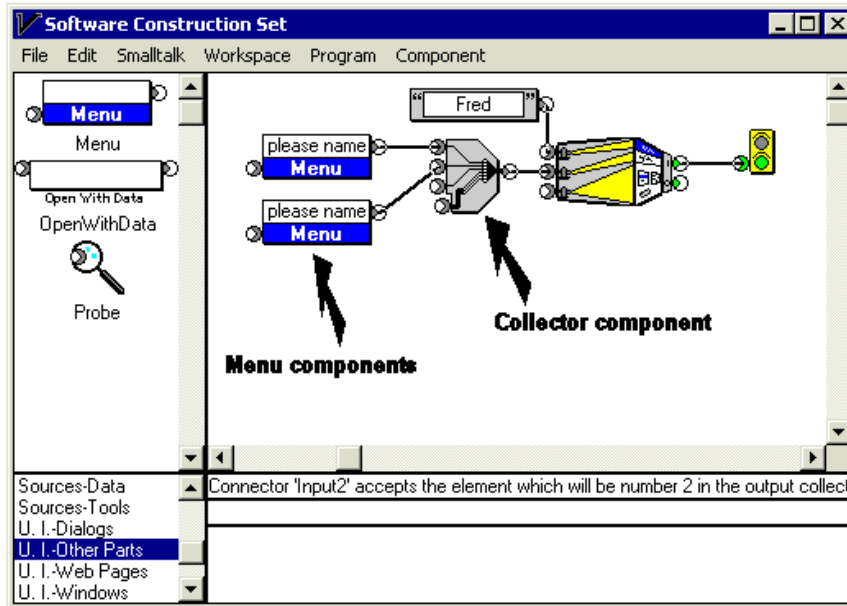


Figure A.6  
Two menus are added to the window

What's new here are two Menu components and a Collector component. The output of the Collector component is fed into the Menu Bar receive connector (the middle receive connector) of the Resizable Window component. Why do you put a Collector component there? A menu bar is a *list* of menus, projected as a horizontal row. (A *list* of things is a collection of these things in a specific 1..2..3.. order.) You will see later that each menu is a list of menu items, *each of which is simply a projection of a command*, projected as a vertical column. By convention, the first menu in the menu bar will be the File menu and the second menu will be the Edit menu. The Menu Bar receive connector of the Resizable Window component is therefore expecting to receive a list of Menu flow objects.

The function of every Collector component is to receive various things and to output a list of these things. The top receive connector will receive the thing that becomes first in the list, and so on. The fourth receive connector is different. It receives a *list* (usually from the output of another Collector component) and tacks this list on to the end of the three-element list it has built. Thus, Collector components can be daisy-chained to build lists of any size. (The icon of the Collector component is meant to suggest wrapping a bunch of wires into a cable.)

## Name the Menus

In Figure A.7 you have typed the names “File” and “Edit” into the Menu components. You see that each Menu component projects into its part of the window’s menu bar.

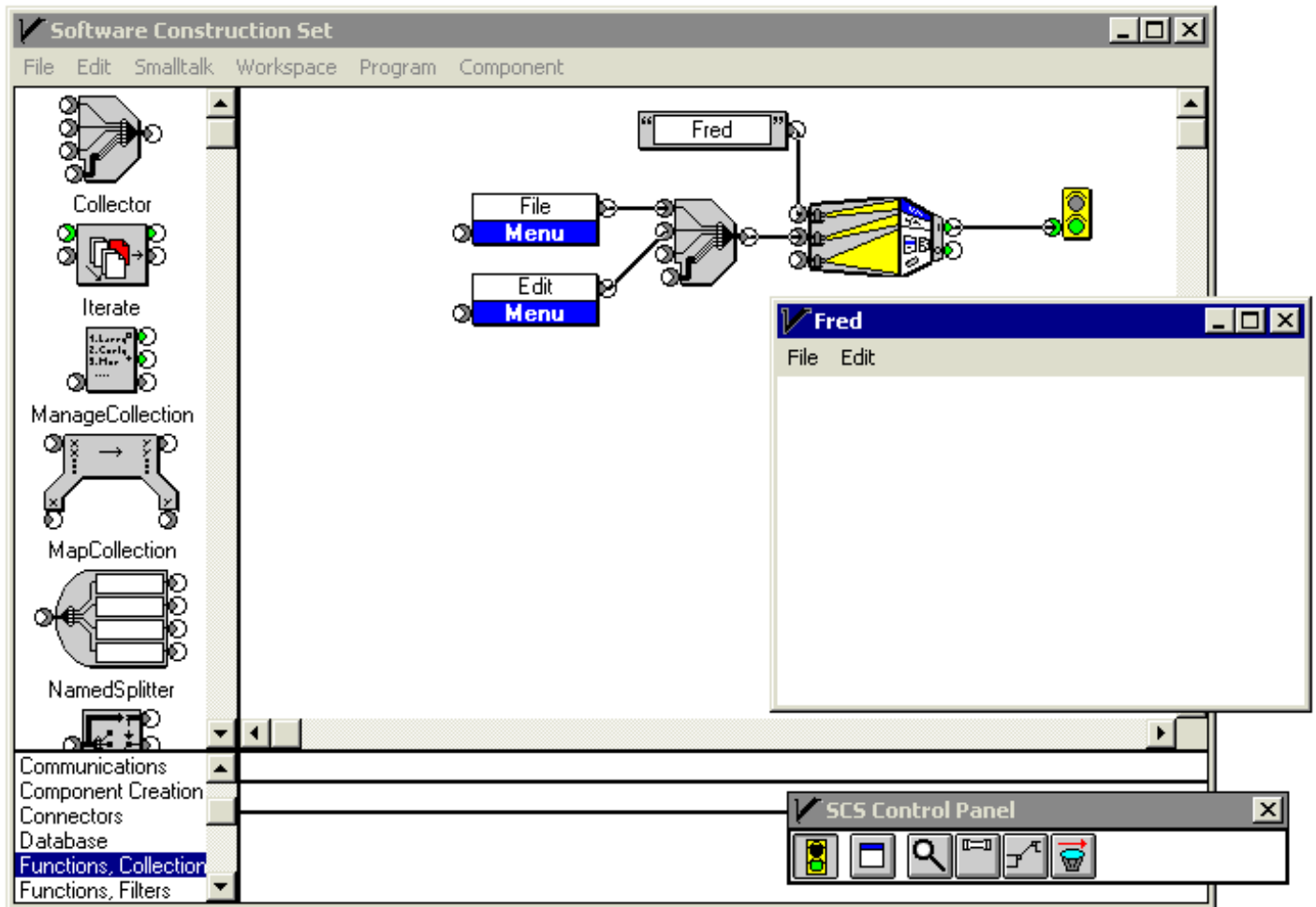


Figure A.7  
Names have been added to the menus

Of course if you click on the File or Edit menu of the application window no menu will drop down because you have not created any menu items (which are projections of commands). That is next.



## Add a Menu Item Placeholder

Here is an example of the useful practice of adding some temporary wiring as a “placeholder” for some wiring which will come later. The “Beep” component has been wired in (through a Collector, because the Menu component expects a *list* of command flow objects) to be projected as the first menu item of the File menu. The Beep component sends a command flow object which causes the computer to beep when its command is picked.

Notice the “Add Label” component, with “Ding” typed into it, between the Beep and the Collector. *Every flow object has the ability to provide its own label.* In the case of a command which is projected as a menu item, the command flow object’s label becomes the text of the menu item. The function of the Add Label component is to pass any input flow object through to its output, meanwhile tacking on to that flow object the typed-in label. You see that “Ding” actually shows up as the first item of the File menu. Clicking that menu item picks Beep’s command and causes the computer to beep.

Finally, notice that the Beep’s command flow object is an example of a flow object that has traveled a long way, touching six components (including Beep) in the process, and finally ending up projected onto the user interface as a menu item.

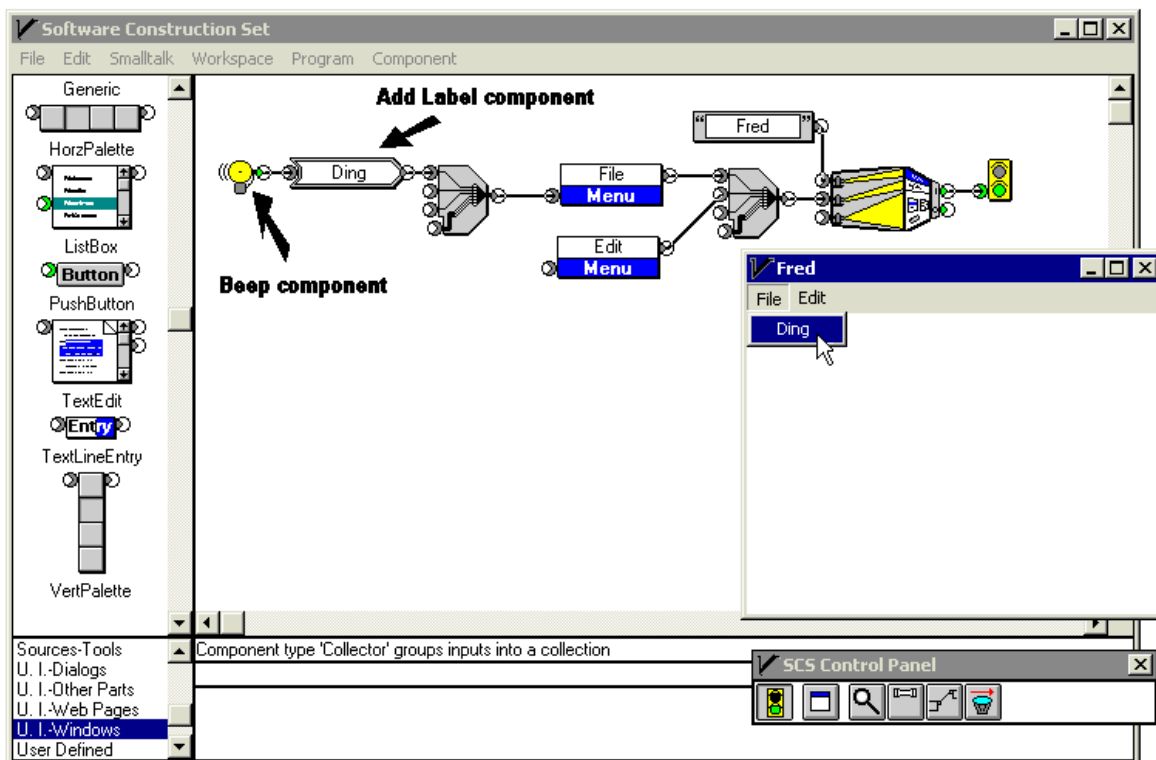
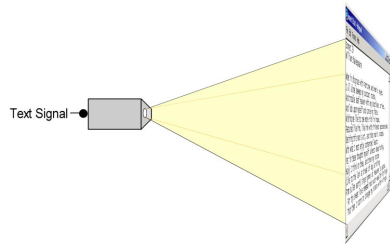



Figure A.8

## Add the Text Edit Component



The meat and potatoes of the application you are building is the Text Edit  component. It has two jobs.

1. It projects a rectangular area of text into the main content area of a window.
2. It outputs a list of several editing commands, which typically go into an “Edit” menu.

The Text Edit component is the component that performs the projection function shown to the left. The Text Edit component’s receive connector (on the left edge) accepts a text flow object; this text flow object supplies the body of text which is projected into the window. In Figure A.9 the bottom send connector of the TextEdit component is wired directly into the bottom receive connector of the Resizable Window component. This is how the Resizable Window component knows to work in tandem with the Text Edit component in the projection of both the border of the window and its content area.

The top send connector of the Text Edit component sends a *list* of standard editing command flow objects which are fed directly into the Edit Menu component. You can see the resulting menu items in the application window.

Again, you wire a temporary placeholder to supply text to the input of the Text Edit component. This placeholder is a Text Source component holding “Now is the time ...”.

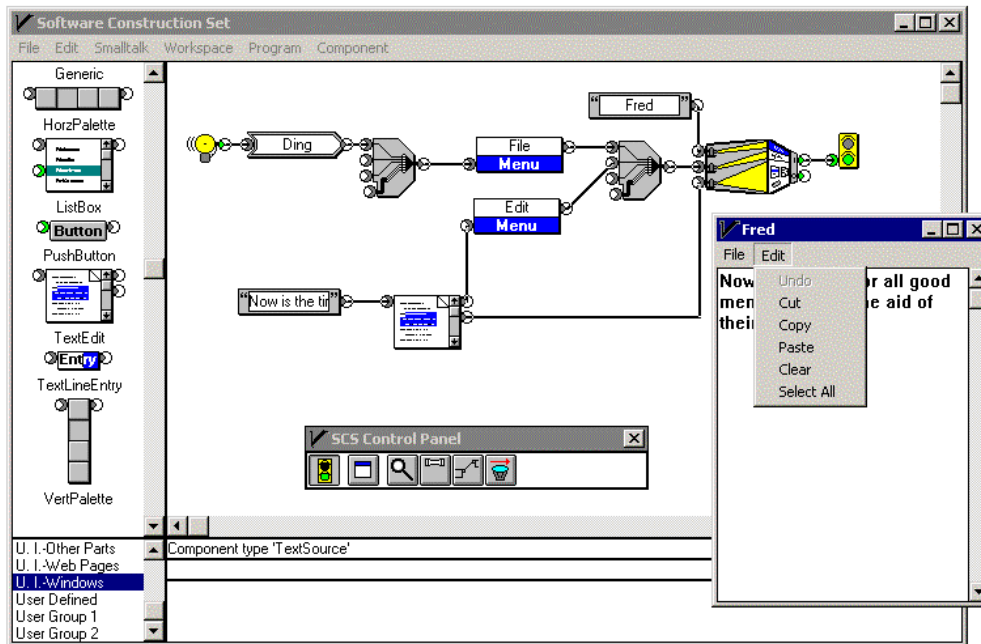


Figure A.9

Addition of the Text Edit component

## Encapsulation

Perhaps you decide that what you have built so far (minus the temporary placeholders) looks like something which other people might profitably use instead of being required to reinvent it. You will now see how the Software Construction Set can *encapsulate* a wiring diagram and turn it into a new component for use in the Software Construction Set.

After reflection, you decide that your new component will be of maximum utility to others if you give these users the ability to determine:

1. the content of the title bar of the window,
2. the list of File menu items, and
3. the text which will be projected into the main content area of the window.

What you must do is find a way to build a new *composite component* with three receive connectors for flow objects whose data determine the values of these three variables. You do this with *connector components*. Figure A.10 shows how you use these connector components. You must assign a name to each connector component to identify it on the outside; here you have used “Title”, “File”, and “Text”.

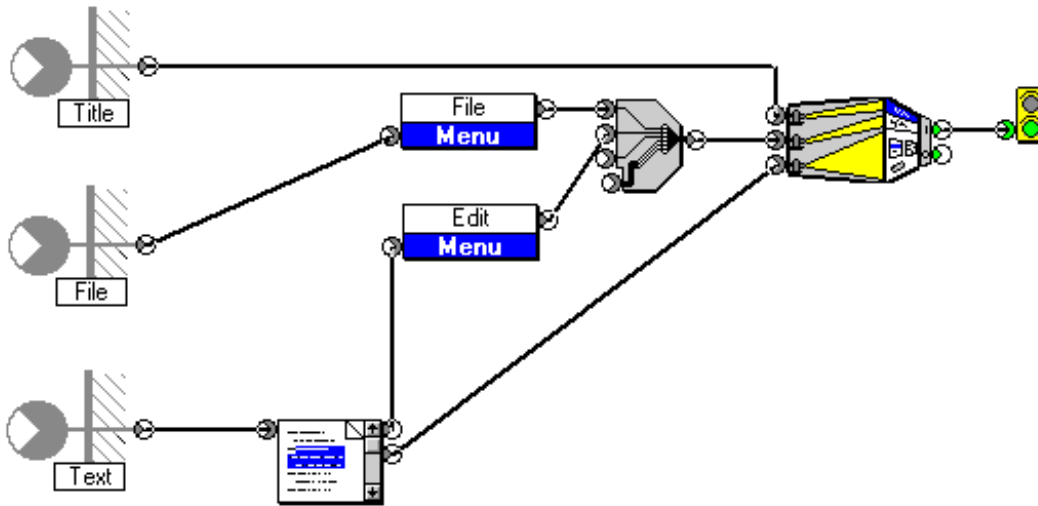


Figure A.10  
Making part of a wiring diagram reusable as a new component

The crosshatched part of the icon of the connector component is meant to suggest the wall of the new component, through which wall the receive connector is connecting the inside of the new component to its outside.

## Create a User-defined Component

After creating the wiring diagram in Figure A.10 above, you select the “Encapsulate” menu item under the “Program” menu of the Software Construction Set. Part of the behavior of this menu item is to ask for the new component’s name; you enter “EditWindow”. After the command completes execution you scroll down to the “User Defined” component palette; there is the new EditWindow component. (The plain look of these composite component icons comes from the fact that they are computer-built.)

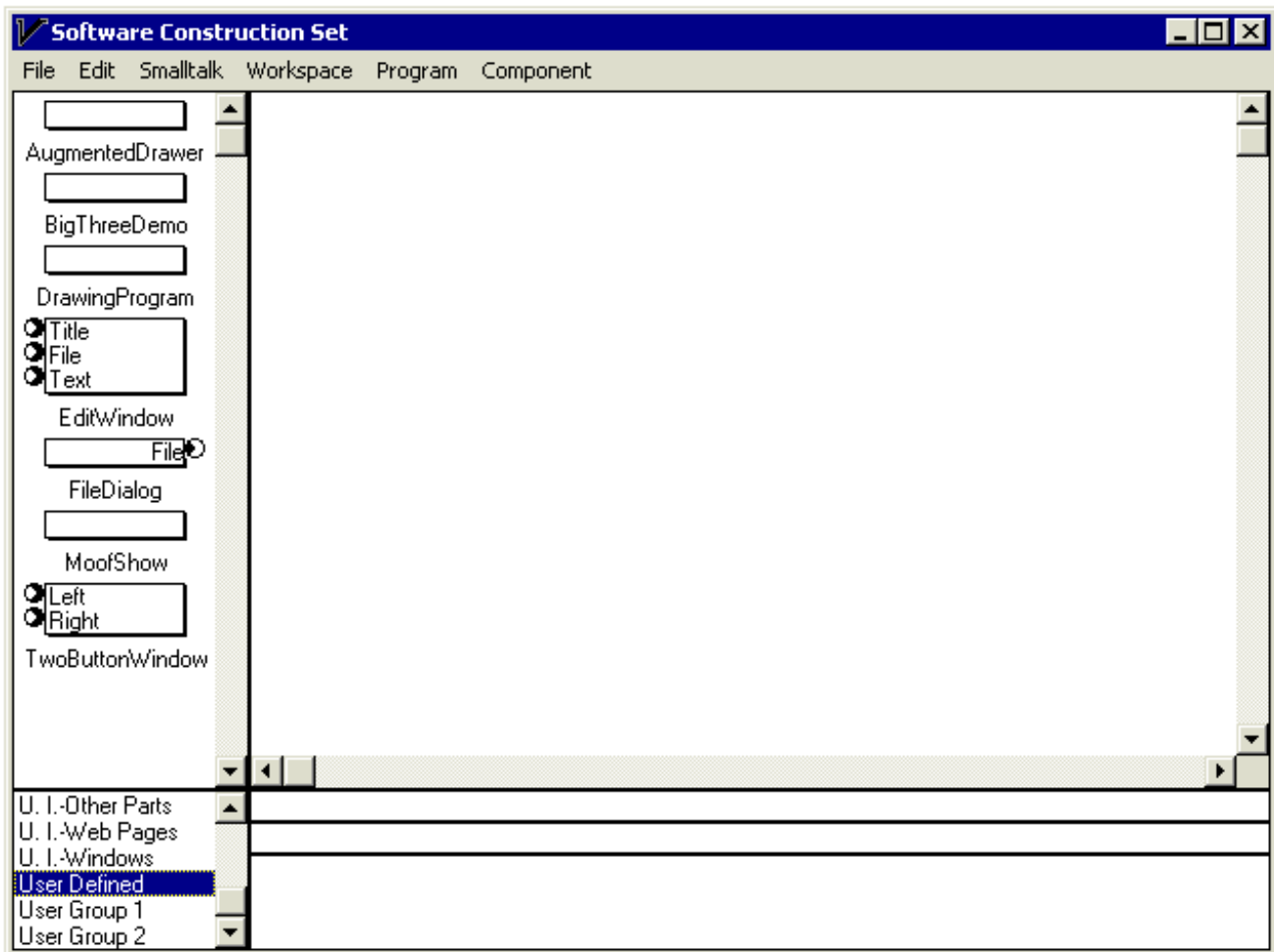
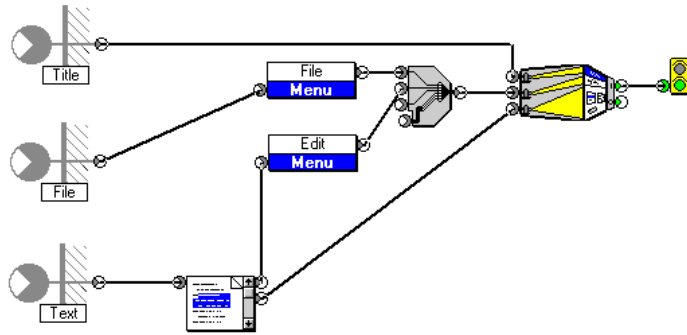


Figure A.11  
The new EditWindow component

## Test the User-defined Component

Now the job is to use the new EditWindow component to continue to build the text editing application. A possible first task is to “smoke-test” the component. (This is a term from the culture of electronic engineering: turn it on and see if it goes up in smoke. Of course it won’t go up in smoke, but it would be nice if it did what we expect it to do.)



Test the component by dragging it into the workspace and wiring some temporary placeholders to it. You see in Figure A.12 that the body of text, the title bar, and the Edit menu are properly projected. The two text flow objects have “gone through the wall” of the new EditWindow component to the wiring diagram which you encapsulated inside it (see left).

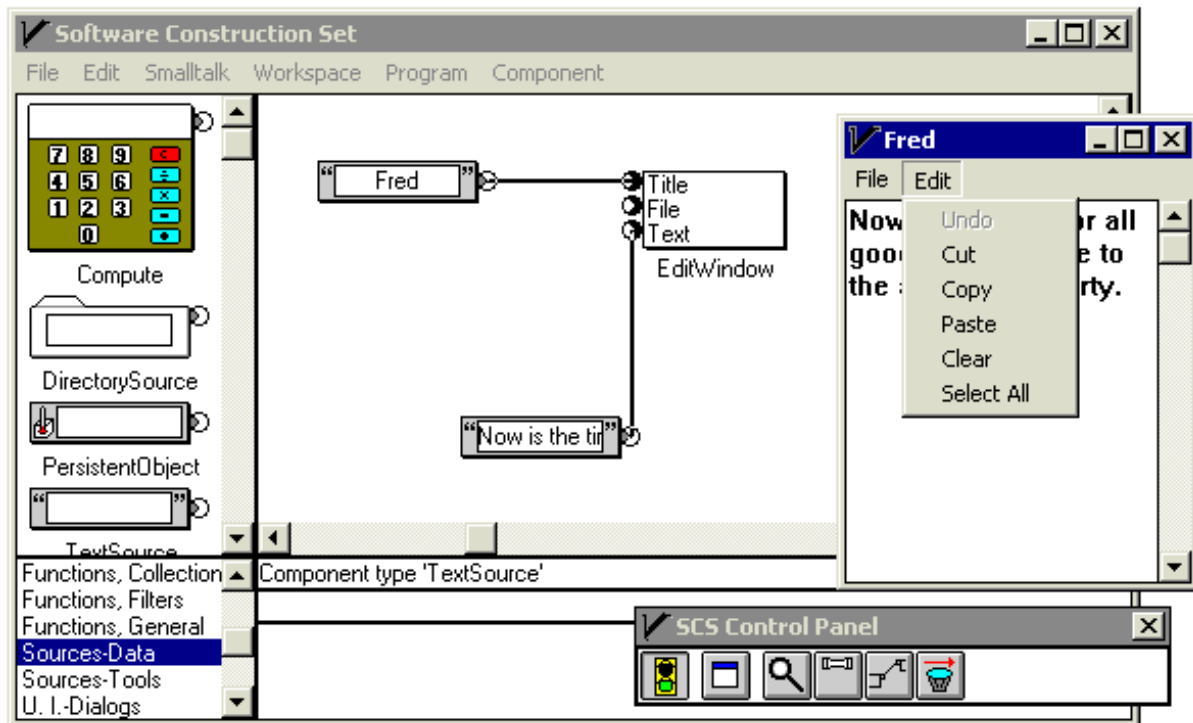


Figure A.12  
Testing the new component

## Open the Text File

The major remaining task is to obtain the text file “Sonnet.txt” and to feed its contents into the “Text” receive connector of the new EditWindow component. To implement the File/Open menu item you must know that the “File” receive connector of the EditWindow component is expecting a list of commands.

First you must build an OpenFile composite component or use an existing one. (This component will be available in a complete product, and it won’t be necessary to build it.) When the command sent by the “Open” send connector of the OpenFile composite component is picked, OpenFile opens a dialog window that allows you to navigate the file hierarchy and find the “Sonnet.txt” file. Figure A.13 shows the application window just before you click the “Open” menu item. (Note that the “Open” label in the menu item derives from the text typed into the AddLabel component, not the name “Open” which you assigned to the top send connector of the OpenFile component when you built it.)

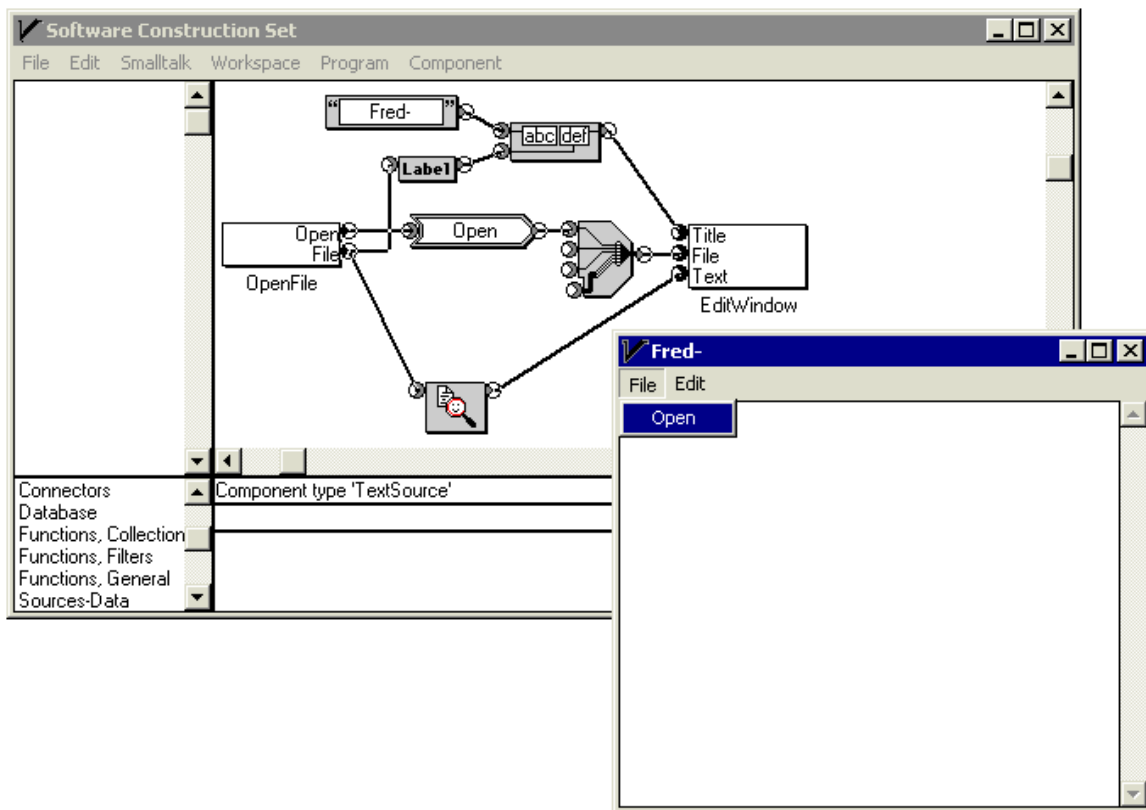


Figure A.13

## Obtain the Text

There are several new aspects to Figure A.14 below, and we'll be considering them one at a time. First, notice that the "File" send connector of the OpenFile component has two wires leading away from it. The way to think about this is that *the same flow object travels down both wires*.<sup>\*</sup> The flow object sent from the "File" send connector contains a file object. A file object is *not* the text contents of the file; it can be thought of as the file name plus the ability to find the file in the file system.

Consider the lower wire out of the "File" send connector of OpenFile; this wire goes to the FileContents component (the one with the magnifying glass in its icon). The FileContents component receives a flow object carrying a file object, finds the file, and sends out a flow object carrying the *contents* of the file (in this case, its text). This output flow object is then wired to the "Text" receive connector of the EditWindow component. Figure A.14 shows that the text has found its way to the application window.

Notice that some text has been selected and the "Copy" menu item is about to be clicked.

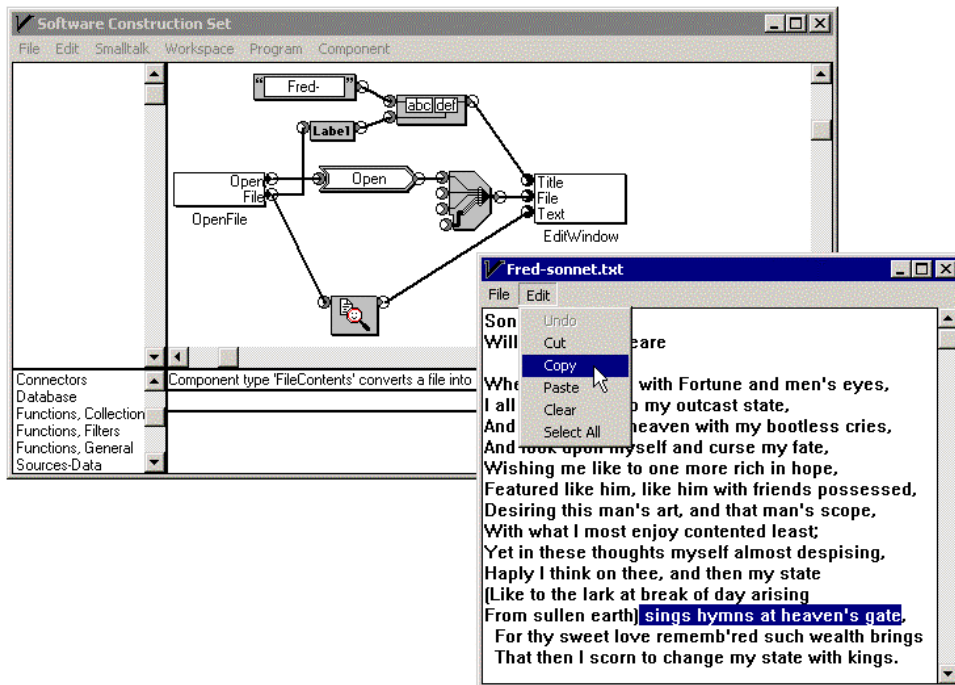


Figure A.14

<sup>\*</sup> If this idea bothers you, think that the flow object is really inside the component from which it originates, and different pointers or references to it travel down different wires.

## Test Copy-Paste

After copying the selected text paste it after the poet's name. You see the result in Figure A.15, which suggests that the Text Edit component is implementing Copy and Paste correctly.

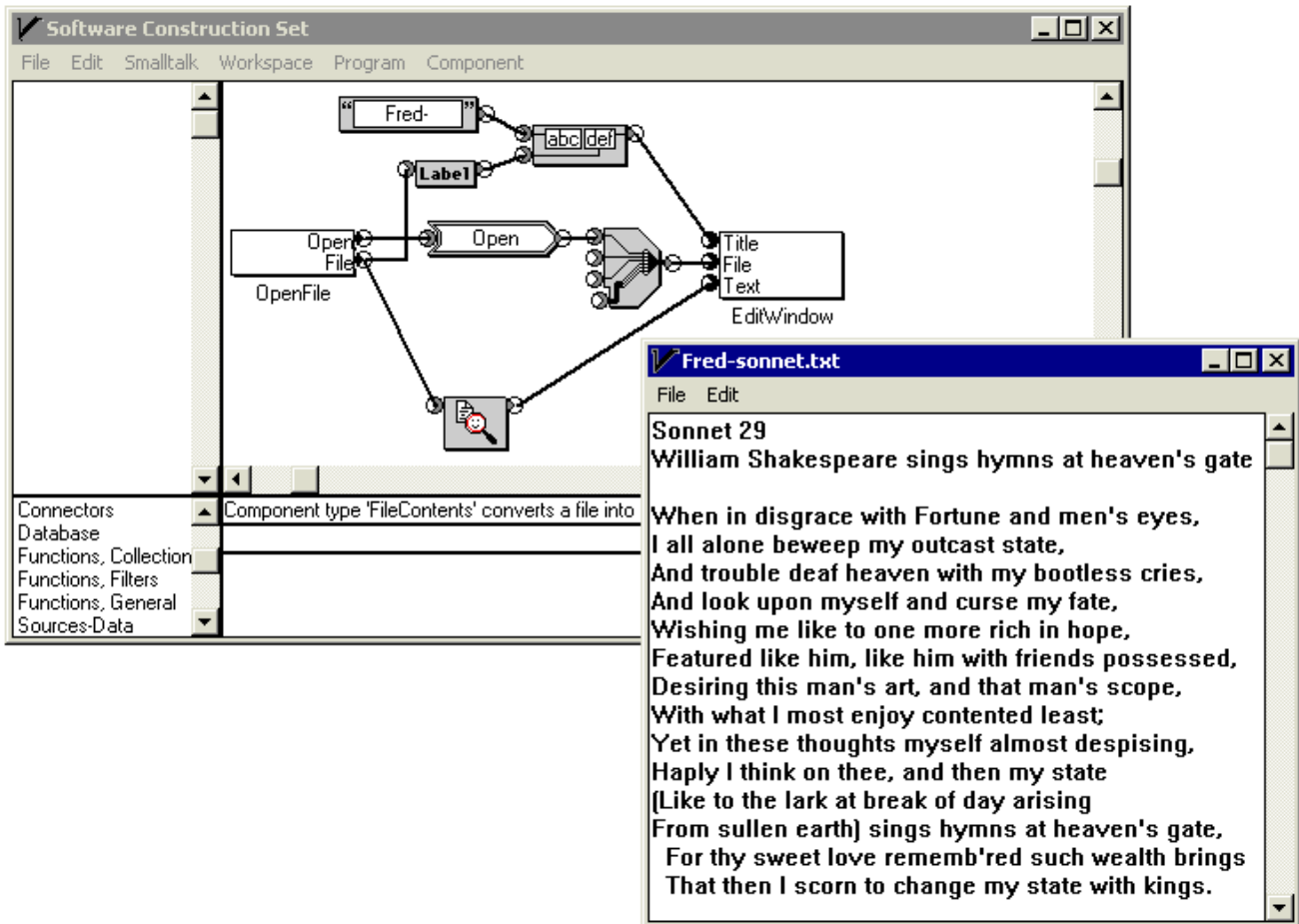


Figure A.15

Now consider the upper wire coming from the "File" send connector of the OpenFile component. This goes to the "Label" component. Recall that every flow object can provide its own label. In the case of a file object flow object, this label is the name of the file (that is, unless this label is overridden by an "Add Label" component). Therefore, the output of the Label component is a text flow object carrying the character sequence "sonnet.txt". This flow object goes to the bottom receive connector of the "Concatenate" component, which tacks together two pieces of text. You can see the output of the Concatenate component in the title bar of the application window.



## Complete the Application

Finally, the application must have a way to exit. The Exit component, with one send connector and a red-light icon, sends out a command flow object with the default label "Exit". When the command is picked, the program exits and the traffic-light button in the Control Panel turns back to red.

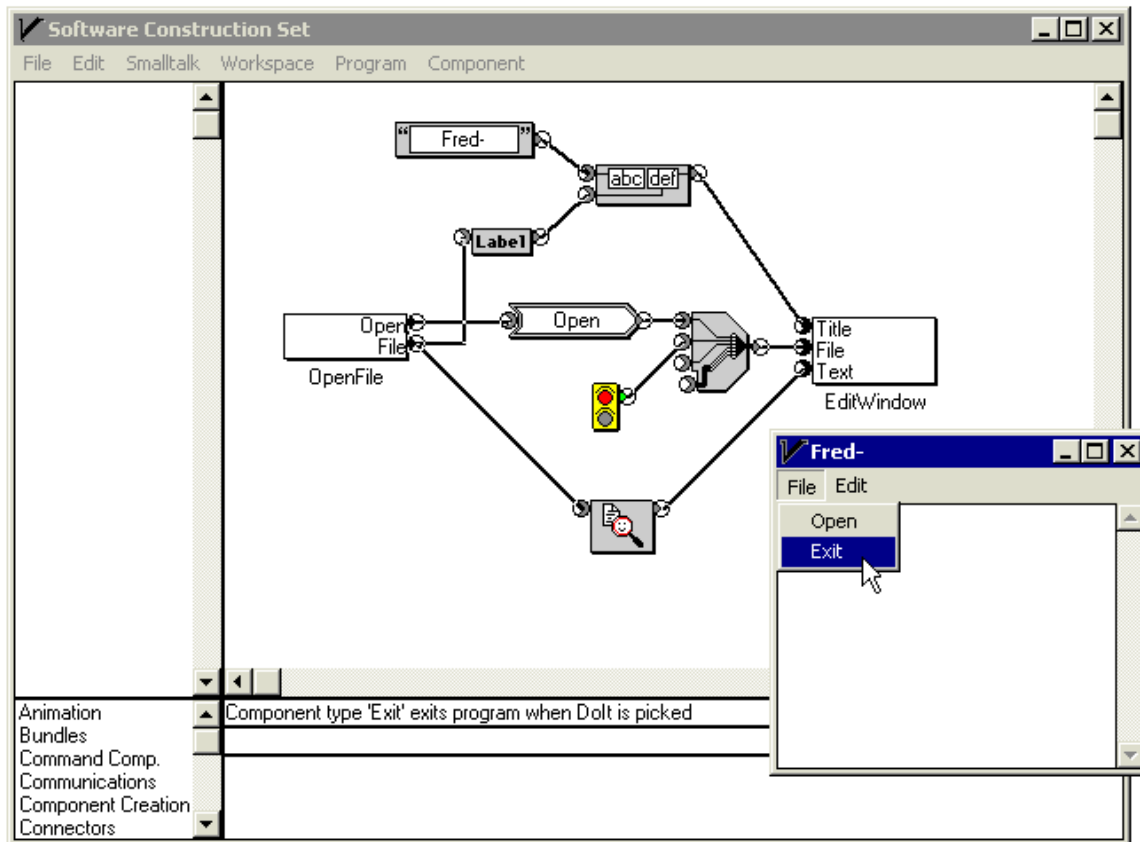


Figure A.16

## Appendix B. The Tight Coupling Between Data and Its Projections

### The Display Update Problem

Every application architecture must have a solution to the problem of updating displays. Let's say that an item of data is projected in two different windows. There must be a provision for it to be automatic that if the application's user changes the data using one window, the projection will be updated in the other window to reflect the new value of the data. Here is a more general statement of the requirement. *It must be possible so that, no matter by what means an item of data becomes changed, all projections of that data item must continually reflect its current value.* The solution to this requirement is called the *update protocol*, and will be described in the next section.

First we shall consider an application which illustrates in several different ways how the results of the update protocol look. The structure of the application is shown in Figure B.1. The window with "Big Three" in the title bar is the application window. The components of the program have been numbered as an aid to the discussion.

The content area of the application window contains a projection of two panes: a text entry line (projected by component 7) and a list box (projected by component 6). We can consider the outputs of components 6 and 7 to be *pane flow objects*; that is, they carry data called *panes*.<sup>\*</sup> Since the content area projects multiple panes, components 6 and 7 feed through a collector component, so a list of panes goes into the bottom receive connector of component 10. The two panes are projected by component 10 in the application window's content area. (We won't discuss here how the positioning of these panes in the window is specified.)

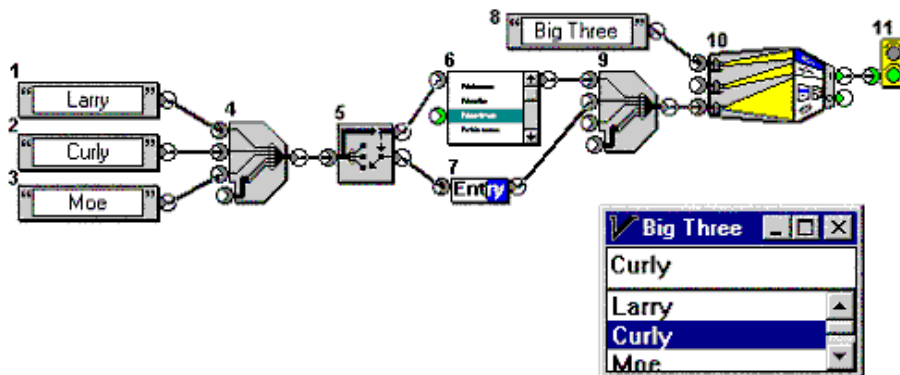
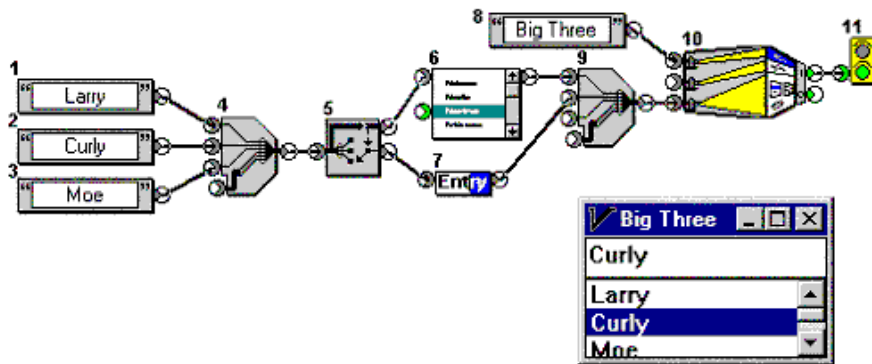


Figure B.1

<sup>\*</sup> There is no need for the pane object to carry the projected data. The data need go no further than the pane component, which projects it.



Components 1, 2, and 3 send text flow objects. The output of component 4 is a flow object which carries a three-element list of flow objects carrying the character strings "Larry," "Curly," and "Moe."

Component 5 (called a selector) dynamically chooses one element of the input list and sends the selected element from its lower send connector.\* The selected flow object carrying one of the three character strings then flows into the text entry line component 7, which projects it into the application window. At the time the screen shot was made, the selector component was selecting the second element, "Curly", of the list.

How did the selector component get told to choose the second element? Note that the selector's top send connector is wired to the list box projector, component 6. A selector component with its top send connector wired to the list receive connector of a component with a choose-one-of-n function (there are several such user-interface components, for example: list box, popup menu, radio button set) is a pattern which appears frequently. The heavy line inside the selector icon, component 5, rising from the receive connector and running across to the top send connector, suggests carrying the whole input list up to the selector's top send connector, from which a flow object carrying the list is carried to the list box component 6, which projects it into the application window, as you see in the figure. The selection position of the selector component and the selection shown in the list box are tightly coupled. When the user clicks in the list box in the application window the selector and the list box components interact behind the scenes so that the selector position always reflects the selection in the list box.\*\*

\* The icon art is meant to suggest a multi-pole switch which selects one wire in a cable and routes the flow object on this wire to the lower send connector of the selector component. This icon is static; the switch will always appear to be on the third wire.

\*\* An incorrect way of doing things would be to feed the selected element out the right side of the list-box component instead of feeding it out of the selector component. This approach comes from the conventional dataflow view but does not incorporate the projection concept. The function of the selector component must be separate from, and tightly coupled to, the user-interface choice function.

## The Update Protocol

**Flow Object Dependents.** For every component, each of its receive connectors is defined to be either subject to the update protocol or not subject to the update protocol. If a receive connector is subject to the update protocol, the receive connector must register itself with each incoming flow object as a *dependent* of the flow object. Then, each time that the flow object or its data undergoes a change, the flow object notifies the receive connector that a change has occurred. The receive connector then notifies its component, giving the component a chance to look at the flow object (and the data the flow object is carrying) to see if the component should re-perform its function based on the changed input. Figure B.2 shows with arrows which receive connectors are subject to the update protocol.

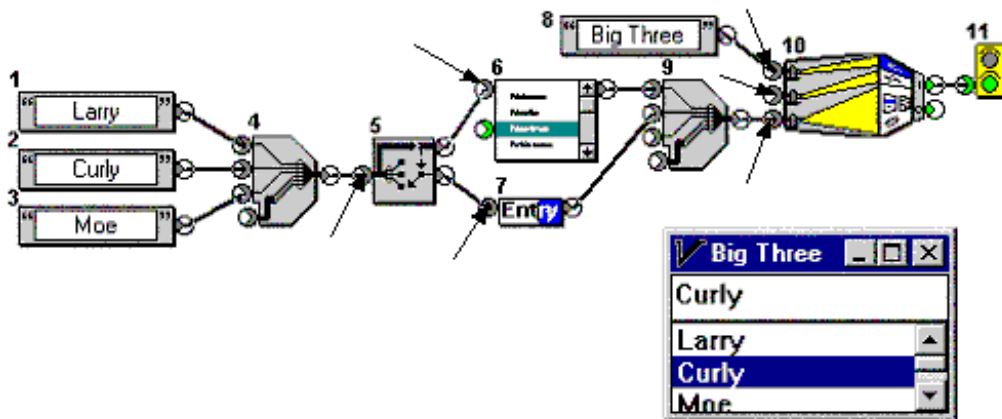
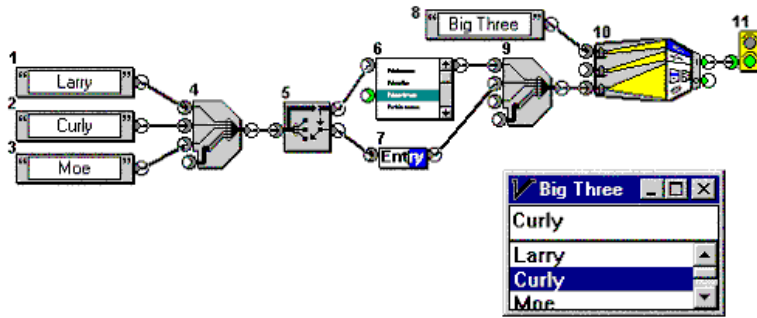


Figure B.2



**How Selection Works.** What we're going to show now is that the tight coupling between the selector component 5 and the list box component 6 is an application of the update protocol. What makes this so is the nature of the flow object that is received by the list box component 6; this flow object is *not*

the same as the flow object received by selector component 5. The flow object received by list box component 6 is carrying a piece of information that the input to selector component 5 is not carrying. What is it?

Let's look at this from the projection perspective. According to the theory, the list box projects the data entering component 6. If you look at the list box projected in the application window you see that there is indeed an additional piece of information: information about the selection, indicated by the highlighting of the "Curly" line.

**Selected List Data Type.** Figure B.3 shows the difference between the "List" data type which enters component 5 and the "Selected List" data type which enters component 6. The "selection" part of the Selected List data item specifies that "Curly" (i.e., number 2 of the list) is selected. This part of the Selected List item is projected in the list box as the highlight on the "Curly" line.

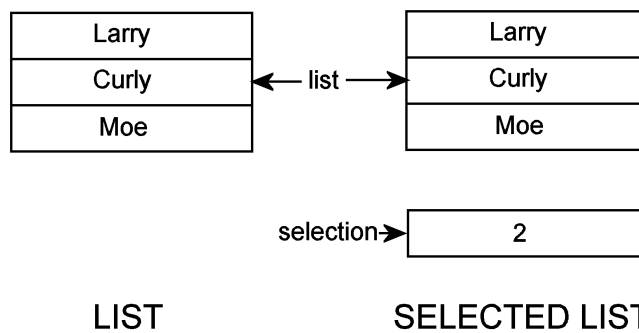
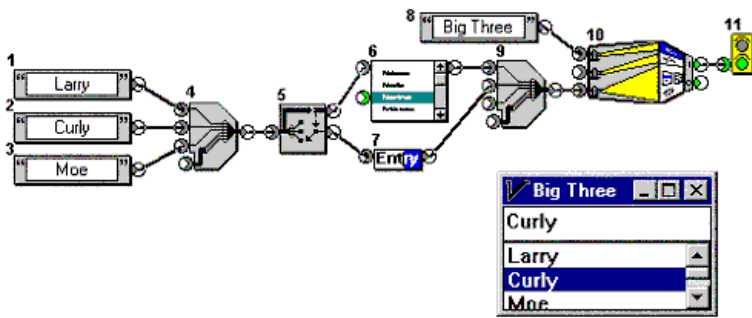


Figure B.3  
Difference between the List and Selected List data types



**Selector State Variable.** Selector component 5 carries an internal state variable which indicates the position of its virtual selector switch; its value, in the case shown, is 2. This selector component constructs the data output (the selected list, which will be carried by a flow object from its top send connector) by combining the input of its receive connector (the list) with this internal state variable (the selection).

**Owner.** Every component that creates a new variable, as the selector component does with its selection variable, is said to *own* that variable. The selection value in the Selected List value sent by selector component 5 is owned by that selector component. Other components, which receive a flow object bearing that variable, may have the ability to change the value of that variable. Any component which changes the value of a variable must notify the variable's owner that a change has occurred.\*

**List Box.** The list box component has the ability to change the selection part of the Selected List value which it receives. It does this in response to an event which comes to it from the user interface. (Notice that the selection variable is in the selector component, and that's where it is changed.) After changing the value of the selection variable, list box component 6 notifies the Selected List flow object, which notifies the selector component 5 of the change.

**Selector.** In response to receiving this notification, the selector component looks at its selection variable and rotates its virtual switch appropriately, selecting the specified element of the input list. It then sends a new flow object carrying that new element out its lower send connector.\*\* Finally, it notifies the flow object, which notifies the dependents of the Selected List flow object to look at their possibly changed inputs.

Notice that the selector component plays no role in changing the value of its selection variable. Also notice that if there were several list boxes wired to the selector component, the selection in all of them would change in response to a user changing the selection of any one of them.

\* You may notice that what is being described here also accounts for how commands work. In other words, commands are nothing exceptional either.

\*\* A new flow object will propagate down the wiring, so the update protocol is not necessary for its dependents to be alerted; its arrival at the receive connector does that.

## What-You-See-Is-What-You-Get Is *Not* an Illusion

Now select "Moe" and remove the final "e" by editing in the text entry line. Here is the result.

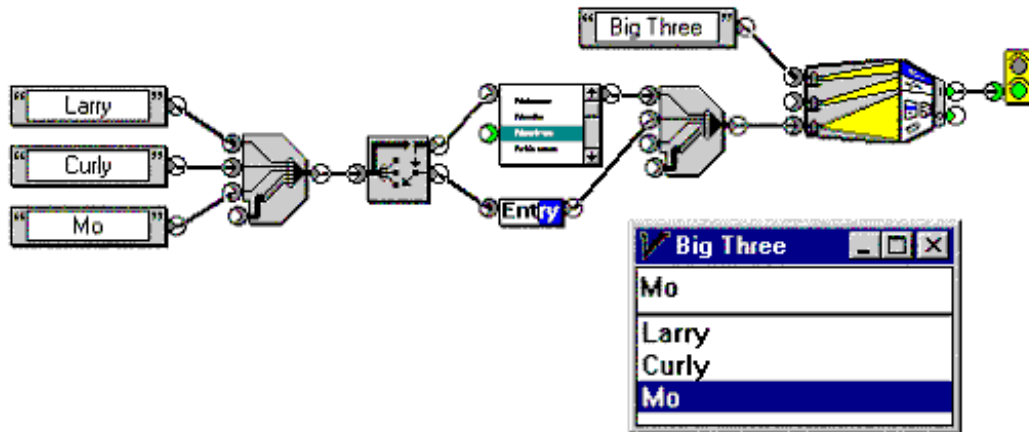


Figure B.4

Looking at the "Big Three" application window in Figure B.4, we see that the change from "Moe" to "Mo" in the text entry line propagates to the corresponding line in the list box. This is because the editing operation directly changes the text where it is *owned*, namely at its original source, the bottom text source component. All dependents of this text value, the list box component and the text line component, are then notified of this change. The former updates its projection; the latter ignores the update because, since it was the one that started the whole sequence, it knows that its projection is already correct. This selective inhibition of re-projection, which avoids spurious flashing of the display, is also part of the update protocol.

Figure B.5 has the same information as Figure B.4, except that it is shown in the context of the Software Construction Set.

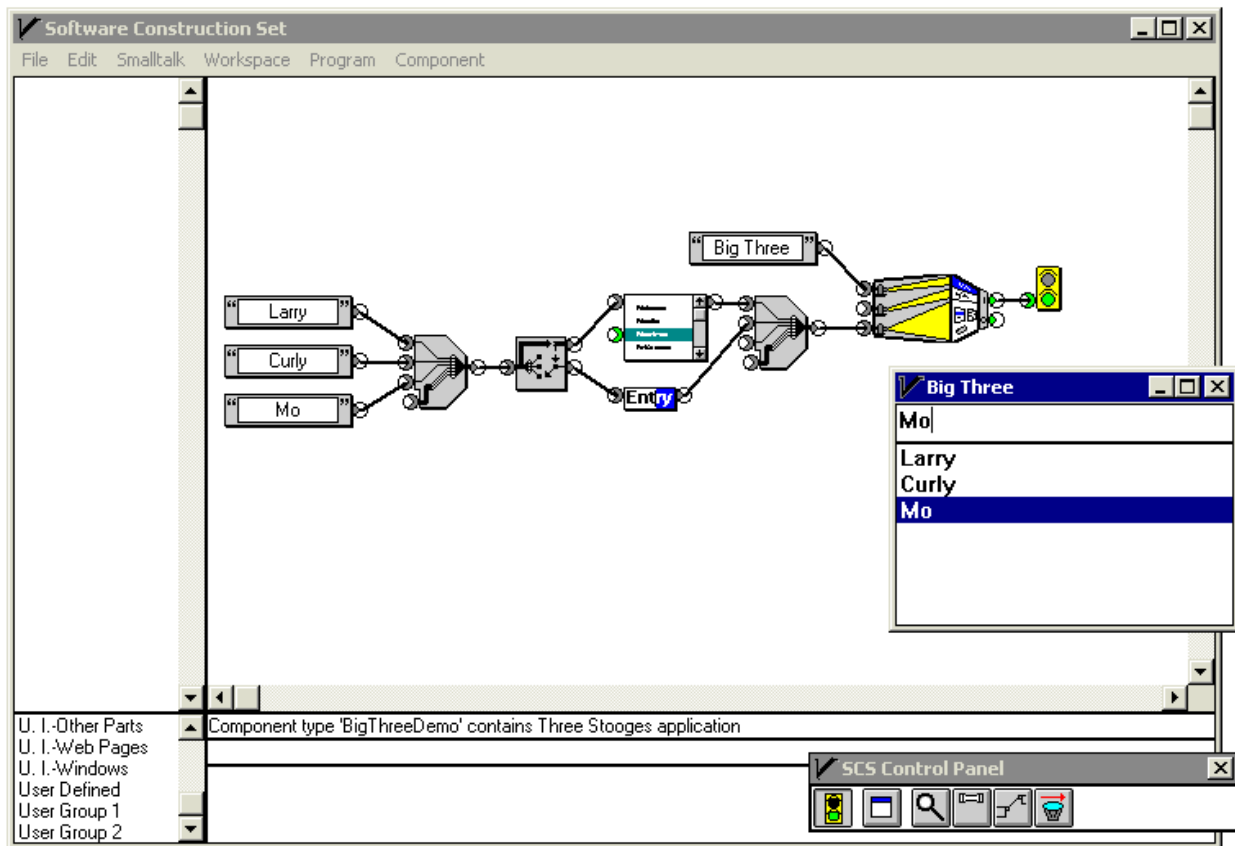


Figure B.5

Look at the bottom text constant component in the wiring diagram in Figure B.5; it now shows "Mo." This occurrence is not in the application window but in the workspace. How did that happen? Let's answer by reviewing what is projecting what.



Let's look at Figure B.6, below.

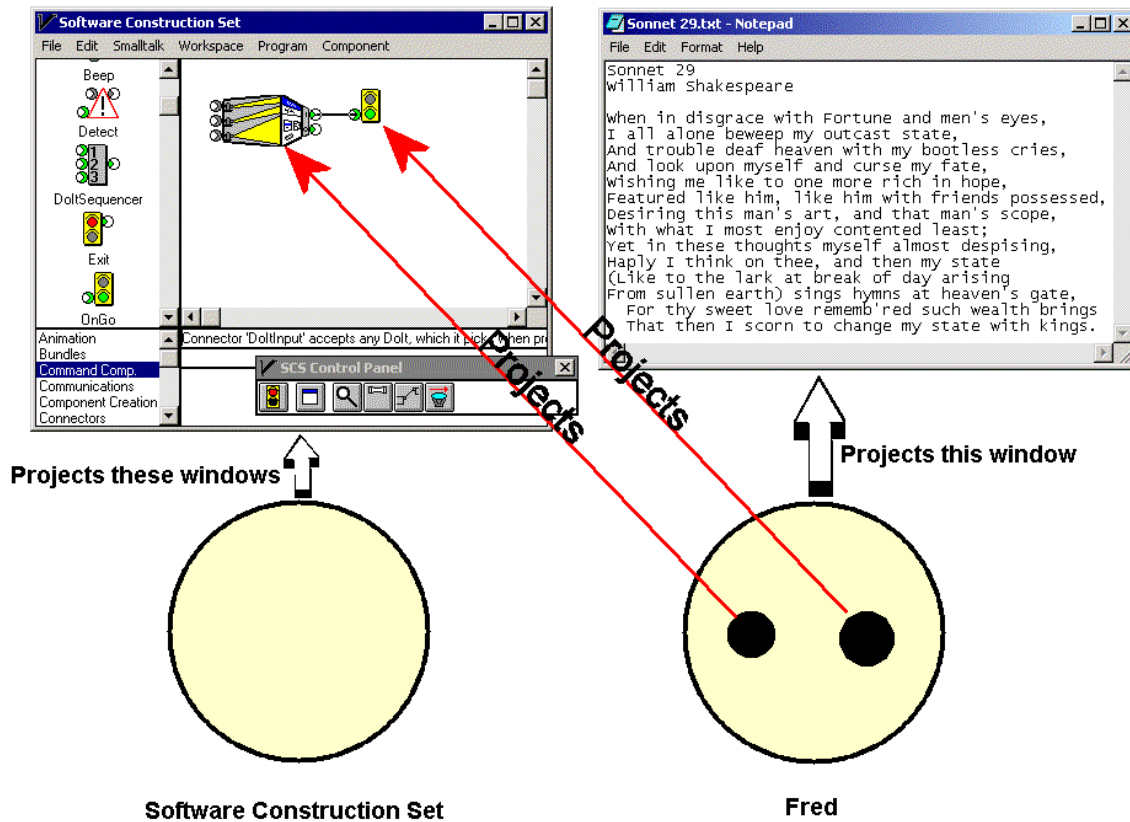


Figure B.6  
The Software Construction Set and the application,  
shown with the windows they project

You know that a function of the application being developed is to project the application window (both are on the right half of the figure). You also know that the wiring diagram is projected onto the workspace, which is a pane of the construction window, and that a function of the Software Construction Set is to project the construction window (both are on the left half of the figure). *But what is the workspace pane in particular a projection of?* Answer: *The internal structure of the application (lower right!).* In other words, What-You-See-Is-What-You-Get is not an illusion. It's how the Software Construction Set works.

## What Is the Software Construction Set?

Figure B.7 is a redrawing of The Software Construction Set and the application with emphasis on the Resizable Window component of the application.

The user's application (the bottom circle on the left, a.k.a. Fred) is a container that contains its components. The software construction set (the top circle) in turn contains the user's application. The function of the Software Construction Set is to project its two windows: the construction window (containing the workspace) and the Control Panel. The Software Construction Set does *not* draw the components inside the workspace.

Each component of the application is an object with two behaviors: the "build" behavior and the "run" behavior. Both of these behaviors are going on at the same time, while the Software Construction Set is operating.

The build behavior of every component is the same; among other things, it projects its icon onto the workspace. That is, the build behavior manifests itself in the workspace.

The run behavior of each component is specific to the component type. The run behavior is the component's part in the behavior of the application. If the component is a user interface component (as the Resizable Window component is) then the run behavior of the component manifests itself in the user interface of *the application*.

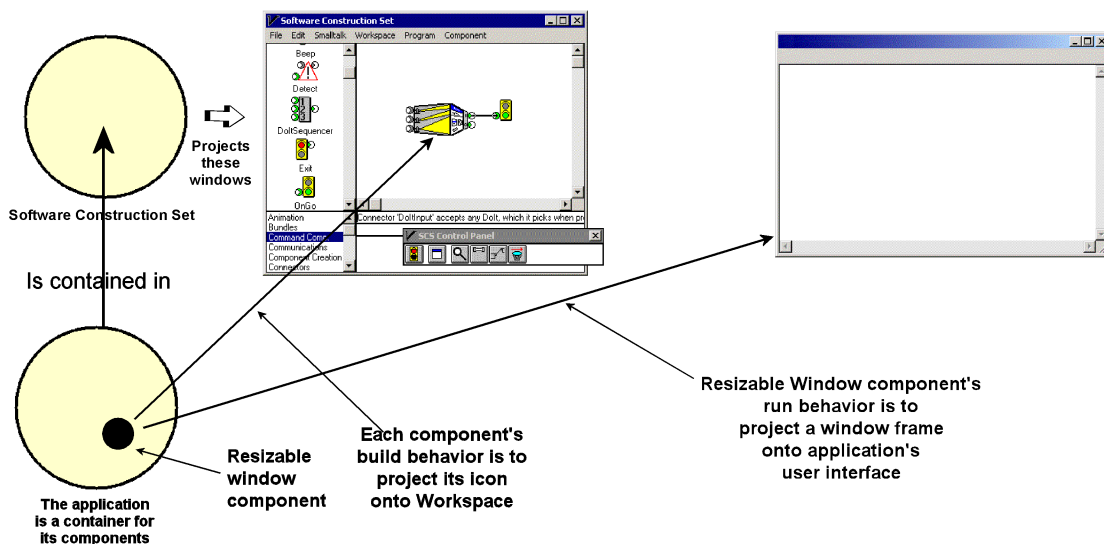


Figure B.7  
The projections happening in the Software Construction Set

When you edit the workspace, you are modifying the structure of the application. The structure of the application is tightly coupled to its projection in the workspace.

If you have programming experience, you see that *this is not your traditional edit-compile-run-debug development cycle*. There is no translation from “source” to “object” language. *The Software Construction Set is a visual editor of an object structure*. This object structure is the application you are developing. The application has its own autonomous behaviors, which include projecting an application window. The behaviors of the application object structure are independent of the behaviors of the Software Construction Set, which is operating on (i.e., modifying) this object structure.