Building Event-Driven Applications With Dataflow Components

Melvin E. Conway

8 Brookhead Ave., Beverly, MA 01915 USA
conw2005@b2bwise.com

Abstract. The synthesis, described here, of a dataflow program execution architecture, a flow-metaphor "wiring" language for representing event-driven interactive programs employing this architecture, and an interactive wiring tool for building these programs has the following properties.
1. The developer assembles programs by wiring up flow components from a library.
2. A multi-level abstraction and reuse mechanism permits wiring diagrams to be encapsulated to form new flow components.
3. The structure of a running program is isomorphic to the structure of the developer's wiring diagram.
4. The interactive wiring tool and the program under development run concurrently, not serially.
5. Display updating and event handling are accomplished by direct messaging, not flows.
6. This architecture suggests a partitioning of the information assets of an enterprise into business objects and flow components, and a partitioning of software technologists into software engineers and more user-responsive application builders.

1 Introduction

1.1 Motivation for the Dataflow Conceptual Model

The file-in, file-out processing model has been with us since punched cards and magnetic tape. Even though it has little relevance to the way applications are built today, the input-process-output conceptual model is powerful because it is visual and lends itself to concrete metaphors. It has been used for design and documentation [1], [2] but, outside of the computational domain, the author knows of no software tools in commercial use for translation of a graphic flow model into a file processing application.

Event-driven applications are essentially different from input-process-output applications. With respect to the use of the flow-network conceptual model for event-driven applications with graphical user interfaces, it is important to distinguish between

1. use of a flow-network model only as a conceptual model for applications that require a translation step for conversion of a flow-network representation to an executable program that is not isomorphic to the flow-network conceptual model, and
2. use of a flow network model as the structure of an executable program.

Fabrik, developed by Dan Ingalls and others, [3], [4] was a significant existence proof for building interactive programs for which the flow-network execution model of a program was isomorphic to its conceptual model.

1.2 Difficulties Using Dataflow for Event-Driven Applications

There are multiple practical difficulties realizing contemporary graphical event-driven applications with runtime programs structured as dataflow networks. The really hard problems arise from the need for bi-directional flows: from database to user interface for presentation and from user interface to database for event handling. Bi-directional flows require many more processing steps than more traditional software designs; this can show up in poor response times. More seriously, there is an adverse impact on program complexity. As an item of data wends its way from database to presentation, moving in and out of collections along the way and possibly following a context-dependent flow path, it must leave a trail so its flow path can be retraced by flows moving in the opposite direction in response to events at the user interface. Conceivably, there can be several such paths through a single flow component.

The author addressed these problems and developed an approach that is documented in a U.S. patent. [5] The approach is a hybrid between flow and direct messaging, in which flows are only unidirectional and only occur at program initialization. The static structure of a program is dataflow, but events are processed and displays are updated without flows.

This paper describes that approach.

2 Execution Architecture and Wiring Language

2.1 Collections and the update protocol

Fig. 1[1] is used to show the handling of collections and the approach to data presentation.

[1] The figures that look like screen shots are actual screen shots with the following possible modifications: some text might have been redrawn to meet publication standards, or callouts might have been added. The development tool from which these screen shots were taken in the mid-1990s was built using a 16-bit Windows version of Digitalk Smalltalk V.

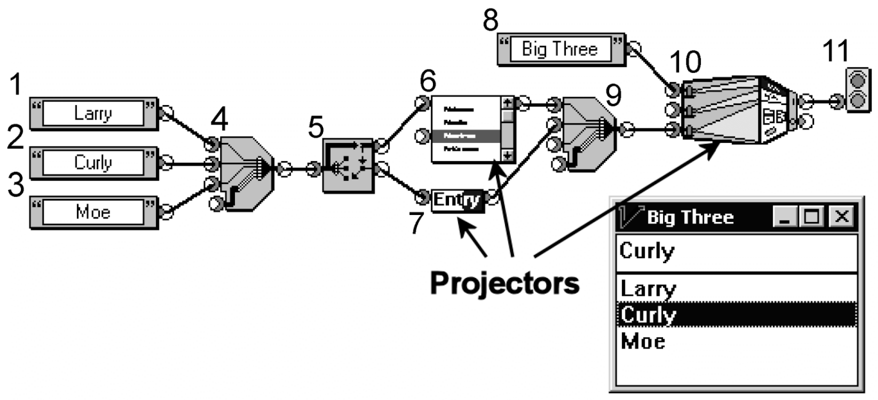


Fig. 1. The upper left part of the figure shows the application's "wiring diagram" under development in the wiring tool's workspace. (Component numbers and callouts were added after the screen shot was captured.) The lower right part of the figure shows the display of the running application. Both views appear simultaneously; the application and the wiring tool run concurrently.

All flows are left-to-right along the wires, from source connectors (on the right edges of components) to sink connectors (on the left edges of components).

First consider components 1 to 7. Components 1, 2, and 3 are Text Source components. Their text-object outputs flow into component 4, which creates a collection containing the three inputs; the visual metaphor is bundling wires into a cable. The collection flows into the Selector component 5, whose visual metaphor is a rotating selector switch. (The component-icon graphics are static.) The Selector component sends the collection out the upper source connector for display by the List Box window pane component 6. This latter component is a *projector*, in that it "projects" the collection information into the list box window pane in the application window at the lower right, and receives mouse events from this list box window pane. The selector switch is tightly coupled to the List Box component; a mouse event that causes the list box selection to change immediately causes "rotation" of the selector switch. The element of the input collection selected by the position of the rotating switch comes out of the lower source connector of the Selector component and enters component 7, which projects the text line window pane into the application window.

Component 10 projects the window frame, comprising the title bar (whose text is received at the top sink connector), the menu bar (not showing here because there is no input to the middle sink connector), and the rectangular client area of the window. The bottom sink connector of the Window Frame component receives a collection (created by component 9) of window-pane projectors, in this case the list box and the text line. (The process by which the panes are positioned in the window client area is not discussed here.)

We shall defer for now a discussion of component 11 and its connection to component 10.

2.2 How This Application Works

Structure of the application. The high-level structure of the executing application is the same as the high-level structure of its wiring diagram. We may therefore discuss operation of an application by referring to its wiring diagram.

Intra-application communication. There are two major types of object in an executing application: components and flow objects. Components (strictly, component classes) are the visible objects in the wiring tool's workspace, such as those shown in Fig.1, that have connectors to which you attach wires to other connectors. A component is instantiated at the time the containing application (which is itself a component) starts and usually lasts until the application ends. Each flow object is instantiated by a component's source connector, which holds a reference to it; this source connector is the flow object's *owner*. Flow objects do not appear on these screen shots, although there are instruments in the wiring tool that make them visible.

In the informal flow metaphor, application data flows down a wire from component to component; some reach the user interface and are projected thereon. This is the conceptual model used to describe the example of Fig. 1. What actually happens is different. Each flow object has a member that references an application data object. Flow objects do not travel down the wires; references to flow objects do. What "travels down a wire" means is that a reference to a flow object is copied from a source connector object to every sink connector object to which it is wired. Thus, neither flow objects nor application data are "moved" or "copied" in the normal operation of an application.

The references to the flow objects available to a component instance reside in the sink and source connector objects of the component.[2] By means of a message through one of its connectors the code specific to the class of that component reaches the flow object referenced by the connector. A component can send a message to the owner of the flow object by means of a message to one of its sink connectors.

The update protocol. The update protocol is a behavior distributed throughout an application and implemented in code inherited by components, connectors, and flow objects. It insulates components from flow objects, application data, and other components. *The update protocol is responsible for the fact that the application behaves as the informal flow metaphor and the program's static structure suggest, without the occurrence of flows.*

Fig. 2 shows the messages involved in the update protocol. In Fig. 2 component A's source connector owns flow object x, which flows through component B to components C and D. The lower source connector of component B owns flow object y, which flows to component E.

[2] The word "instance" is implied if it is absent after the words "component", "connector", and "flow object". Also, "connector instance" is implied after "sink" and "source".

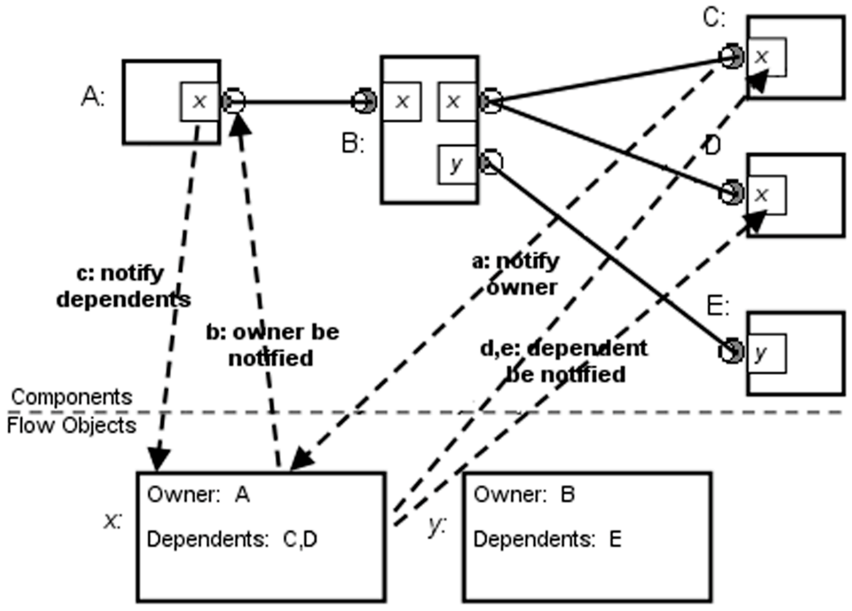


Fig. 2. The sequence of four messages that constitute the update protocol.

Such flows occur, with rare exceptions, only at the initiation of an application, during which each source connector instantiates its flow object and flows (a reference to) it down the wire(s) connected to it in order to establish the connectivity relationships among the components. After that no more flows normally occur and the behavior of an application is determined by the update protocol plus the specific behaviors inherent to components and application data objects in response to events.

When, during this initial series of flows, (a reference to) a flow object reaches a sink connector the sink might tell the flow object that the sink is a *dependent* of the flow object. Declaring dependency assures that the sink will be notified whenever the state of certain members of the flow object or its referenced application data object change. Each flow object maintains a collection of (references to) its dependents.

The update protocol comprises the following four messages that occur in response to an event.

1. Message a: `notify owner`. A component receives an event, alters its own state and the state of the application data referenced by the flow object, and sends a `notify owner` message to the flow object (via a sink connector). This event-receiving component modifies the application data (or any data referenced by any flow object at one of its sinks) in response to the event. Before step 3 below begins, the application data has already been altered.
2. Message b: `owner be notified`. When it instantiates a flow object, the owner of the flow object gives the flow object a reference to the owner and the flow ob-

ject stores this reference. In direct response to the `notify owner` message the flow object simply sends an `owner be notified` message to this owner.

3. Message c: `notify dependents`. The component whose source connector owns the flow object has an opportunity to perform application-specific processing determined by its class after receiving the `owner be notified` message. Then the owner connector sends the `notify dependents` message back to the owned flow object.
4. Messages d, e, etc: `dependent be notified`. In response to the `notify dependents` message the flow object sends a `dependent be notified` to each dependent named in its list of dependents.

Application of the Update Protocol to the Selector Component. The tight coupling between components 5 and 6 in Fig. 1 is a common application of the update protocol. It was stated above that the Selector component passes the collection it receives at its sink to the List Box component. This is not completely accurate. Consider the object that the list box is projecting in Fig. 1. It is not only the collection (Larry, Curly, Moe) but something that tells the user that Curly has been selected. The object that is being projected by the list box is not a list but a *selected list*. The difference between a list and a selected list is shown in Fig. 3.

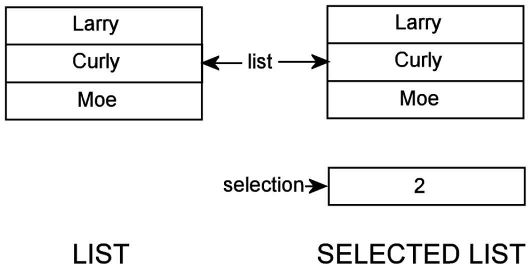


Fig. 3. The list is the input to the Selector component of Fig. 1. The selected list is what is projected by the List Box component.

Consider what happens if Moe is clicked in the list box. The List Box component 6 receives the event with a parameter 3, denoting Moe.[3] The List Box component then modifies the selection member of the selected list data object referenced by the flow object referenced by the list box's upper sink connector by sending a message to itself such as the following (as it might appear in Smalltalk):

```
getSink1 getFlowObject getDataObject changeIndexTo:3 .
```

Clearly every selected list object must be able to act on a `changeIndexTo:` message.

At this point the List Box component has changed the index; it then notifies the Selector component that sent it the flow object to act on the new value. The List Box component does this by invoking the update protocol.

[3] Counting starts with 1 here, as in Smalltalk.

Upon receiving the `owner be notified` message the Selector component knows that its index has been changed, so it changes the data object reference in the flow object owned by the lower source connector and has this connector send a `notify dependents` message to this flow object. Notice that there is no flow.

2.3 Buttons and Menus

A commonplace interpretation of the behavior of a flow component that projects a button on the user interface is that, when the component projecting the button receives a mouse-click event, a right-to-left flow is initiated.

There are no right-to-left flows in this design. Handling of an event at a button or menu item is readily treated with the update protocol.

We define a type of application data object called a *command*. A user-interface component that accounts for a button or a menu item is a projector of a command object. The function of a command object is to invoke the process determined by the command's owner.

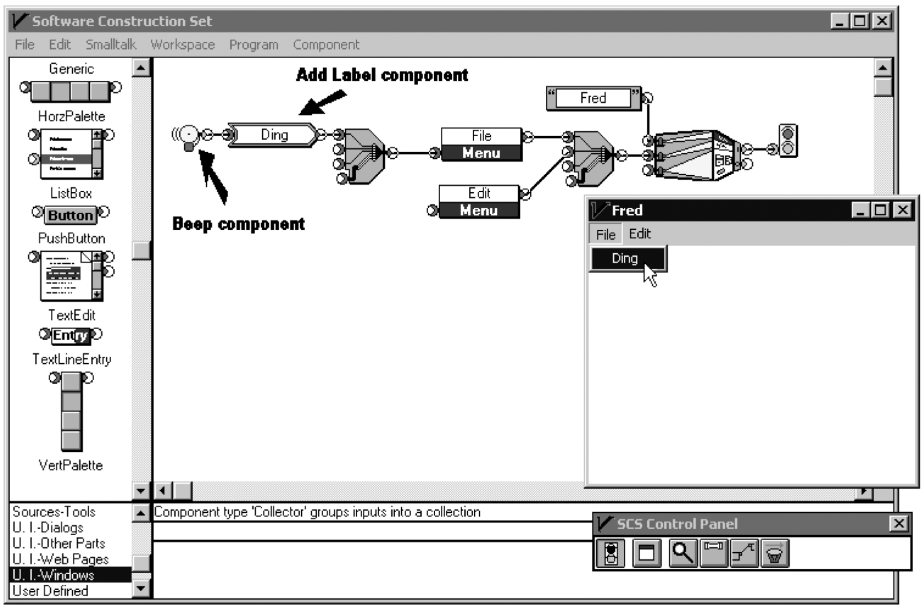
Fig. 4 illustrates the behavior of a command that is projected onto a menu item.

In the informal flow metaphor the Beep component emits a beep command that makes its way out to the "Ding" menu item, which becomes visible when the menu is opened. The actual behavior, described next, is not so different.

To the developer the Beep component has a particular behavior, which is to invoke the system's beep service. The Beep component's source connector emits a flow object that references a beep command data object (for convenience only we call this a "command flow object"), both owned by the source connector. When the command flow object makes its way through the Add Label component it acquires the label "Ding". This label overrides any label with which the beep command itself might respond. A collection containing (in this case) only this flow object enters the Menu component. Upon receiving this collection of command flow objects the Menu component declares itself to be the dependent of every flow object in the collection. In Fig. 4 the one menu item in the projected menu is the projection of the beep command, with the label "Ding". The Menu component's source connector then sources its own command flow object to the Window Frame component, which works with the resident window management software to project an opened menu when the menu's title "File" is clicked.

When the mouse is released on the "Ding" menu item the Window Frame component is notified by the window management software with a parameter indicating that the "Ding" menu item was the one chosen. This causes the Menu component's source connector to receive an `owner be notified` message with this parameter. The Menu component finds the corresponding flow object in its input collection and sends it a `notify owner` message. The Beep component's source connector then receives an `owner be notified` message, which causes it to send an `invoke` message to the command data object. (This sequence of steps, from the mouse event to the `invoke` message is called "picking" the command object.) Every command data object can receive an `invoke` message; its response is to invoke the particular function for which it is responsible, in this case the system's beep service. After the return from this service call the source connector sends a `notify dependents` message to its flow object. When the `dependent be notified` message is received by the Menu component (because it has declared itself to be the flow object's dependent) the Menu component sends a `notify dependents` message to the command flow object of its source connector. Arrival of the `dependent be notified` message by the Window Frame component's sink connector completes the closure of the user-interface menu projection.

We can now consider the connection of components 10 and 11 in Fig. 1. The two source connectors at the right edge of the Window Frame component emit command flow objects that cause the window to open (top source) and close (bottom source). Component 11, called an OnGo component, causes the window to open.



Component type 'Collector' groups inputs into a collection

Fig. 4. The program beeps when the "Ding" menu item receives a mouse click. The wiring diagram and the application window are both shown in the context of the wiring tool.

In Fig. 4 notice the little floating window at the lower right labeled "SCS Control Panel". The leftmost button in this panel is the stop-start button. The icon on the button is a traffic light that is red when the application is not processing events and green when it is. Clicking this button when it is red initializes the application and then causes component 11 to pick the command flow object at its input, opening the window. The button keeps a list of all the OnGo components in the application. After signaling them all it turns its traffic-light icon green.

2.4 Run-Time Scheduling

These descriptions read as though some of these messages were asynchronous. The reader might question the sequence in which these messages occur.

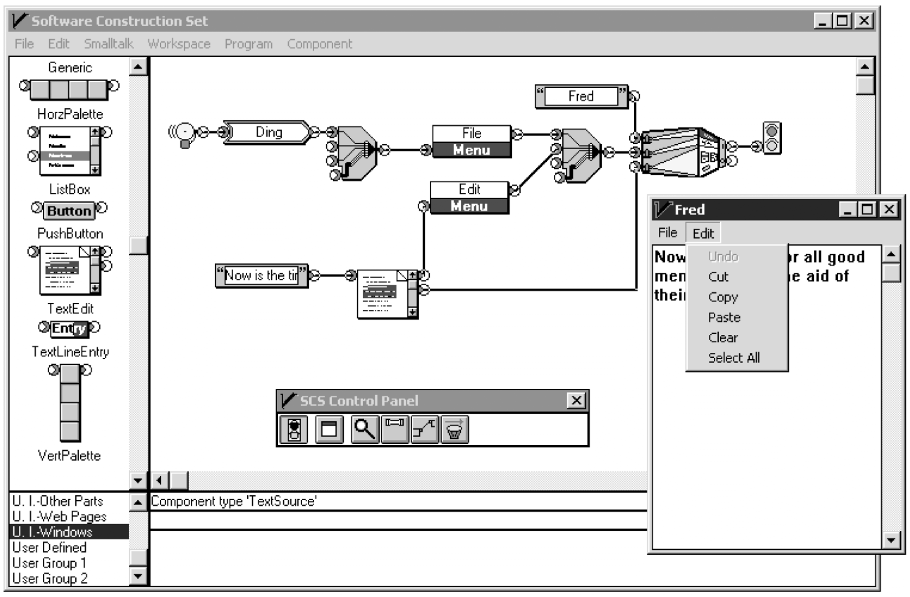
Another scheduling consideration is that a multi-sink components does not know how many of its inputs will change, and it is uneconomic for such a component to respond immediately to every flow or `dependent be notified` it receives.

The author's approach to both of these issues has been to defer processing of any change message arriving at a sink connector; rather, the sink is put into a pending state and the component is queued. Control returns to the queue manager, which dequeues the first-in pending component for resumption of processing. When the queue is empty the processing of an event is complete.

2.5 Abstraction

It is possible to encapsulate a wiring diagram, turn it into a component, and place the component into the library. For this to be useful we need the equivalent of formal parameters. This function is provided by Connector components.

We can cause the wiring diagram of Fig. 4 to evolve into a simple reusable text editor component by adding a TextEdit component from the library as shown at the bottom of the wiring diagram in Fig. 5.



Component type 'TextSource'

Now ... r all good men ... e aid of thei

Fig. 5. An edit component from the library is added to the beep program.

The upper source connector of the TextEdit component emits a collection of command flow objects that implement the standard set of edit-menu items. The component projects the text window pane shown in the figure. The Text Source component feeds the "Now is the time…" string into the TextEdit component.

Now let us remove the beep logic and prepare to encapsulate this diagram into a component with three sinks that accept the title bar text, the command collection for the File menu, and the text string to be edited. We add three Connector components as in Fig. 6. (The vertical crosshatch in the Connector component's icon is meant to suggest the enclosure that will be built around the wiring diagram.) After this wiring diagram is encapsulated it will appear in the library as a component with three sinks with the names shown.

Notice that, because (1)flow objects are what pass through sink and source connectors no matter how many levels deep their components are, and (2)the application data objects can be complex, there is no limit in principle to the number of levels of abstraction that can be useful. Components can be wired in the wiring tool without regard for whether they are primitive or manufactured by encapsulation. Indeed, as is the case with Smalltalk, it is likely that in a practical system even those components that are presented to the developer as part of a primitive set might have inside them not code but wiring diagrams.

If the developer of a component permits, the wiring tool can zoom into the component and display its interior and, again if the developer permits, modify it.[4]

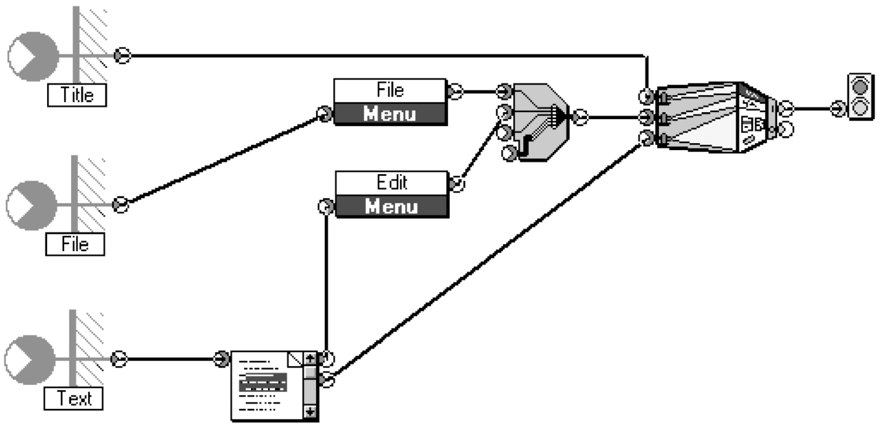


Fig. 6. Three Sink Connector components are added in preparation for encapsulation.

3 The Wiring Tool and the Application Under Development as a Single Program

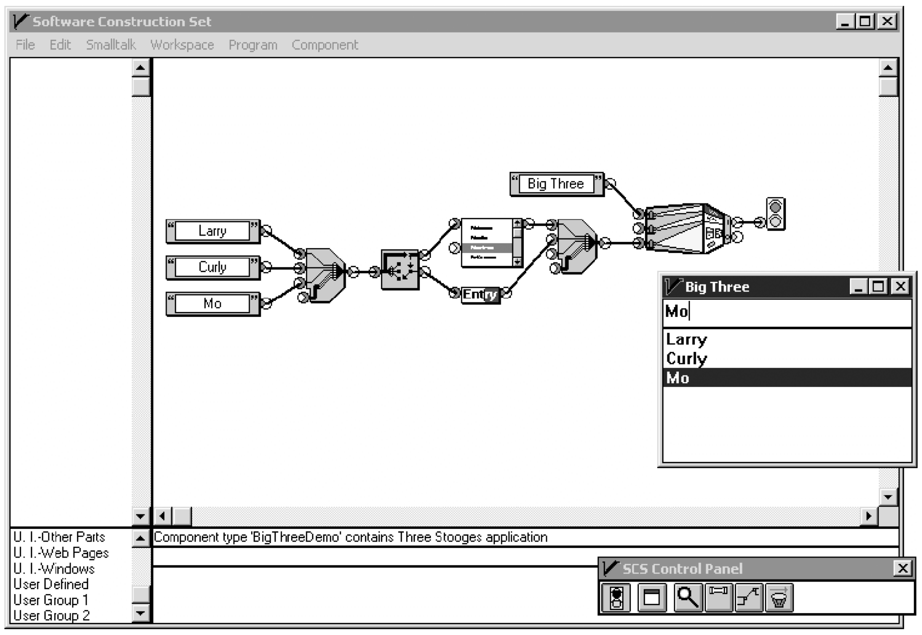
Please take another look at Fig. 1 because we are going to perform an experiment with it. We are going to click on "Moe" in the application window's list box and, in

[4] The permissions granted to the wiring tool by a component depend on how the component is licensed. There is a discussion in the patent of a technological response to this set of issues.

the text line window pane, edit “Moe” by removing the last letter so it becomes “Mo”. Fig. 7 shows the result.

Notice that “Moe” has become “Mo” not only in the text line window pane but also in the list box. That is because there is only one string being projected: the string owned by the Text Source component; both the list box and the text line are projections of that “Mo” string. Indeed, “Moe” has become “Mo” in the projection of the Text Source component icon *in the wiring workspace*.

This behavior is best understood by focusing on the wiring tool rather than the program under development. Think of the wiring tool as a graphical editor that edits an internal structure and maintains an isomorphism between the internal structure and its projection on the workspace. This internal structure happens to be a program with its own behaviors, one of which is to display the application window(s). These two programs—the editor and the program under development—are peers, each responding to its own user-interface events. Besides maintaining the isomorphism between the internal structure and its presentation, the editor responds to removing or adding a wire or component by imposing a flow or event on the program under development so as to maintain the integrity of the flow objects and their references.



Mo

Larry
Curly
Mo

Component type 'BigThreeDemo' contains Three Stooges application

Fig. 7. The application of Fig. 1 in the wiring tool’s workspace after selecting “Moe” and editing the text line by removing the last letter.

Thus, the program may be modified and will continue to behave appropriately. In terms of the user’s experience, it is similar to operating a contemporary word processor: what you see is what you get, without delay.

4 Reuse

The update protocol is an effective firewall, and the principal sources of coupling (and therefore the principal obstacles to reuse) will be the public message sets of the application data objects. There is already much experience designing such public message sets for reuse.

Following is the author's concept of how the use of this architecture might evolve in an enterprise environment. The architecture will induce on the data assets of the enterprise a partitioning into two distinct types: business objects and flow components.

Business objects are the persistent data assets of the enterprise; they reside in databases and are accompanied by their own behaviors, which embody the business rules of the enterprise. *The things that have been called "applications" in this paper are essentially viewers and editors of business objects.* These applications are built by wiring up flow components. Some applications will have long lives and will evolve, and some will be built for one use.

The architecture similarly [6] contains within it a basis for dividing responsibility in the enterprise between two classes of technologist. Business objects, the primitive set of flow components, and the assembly tool(s) will be built and will evolve under the control of software engineers. It is conceivable, perhaps even reasonable, that the libraries of manufactured component classes would be under the control of people who are more aligned with the users of the applications, who are less subject to the strictures of software engineering, and who are therefore in a position to be more responsive to short-term and changing demands from users.

Components interact with business objects by sending messages to them. The author visualizes a messaging interface presented to flow components by business objects as follows. Messaging is dynamic; there should not be the need to rebuild a component as business objects evolve, as long as a business object continues to honor the messages sent by the components that communicate with it. An application builder can query the current set of public messages of a business object, possibly with a pop-up list, and select from this set. Each public message will come with a help facility and a template for parameters. Each parameter value must be locally available within the component. There will be a set of messages inherited by all business objects to support this interface, which will largely be implemented in the wiring tool.

The author believes that this message interface, indeed the entire wiring tool, can be built from flow components.

References

1. "HIPO - A Design Aid and Documentation Technique" [Hierarchy Input-Process-Output], IBM Publication #GC20-1851, May 1975. (Out of print) This citation is indirect; a more direct citation can be found on the Web at http://www.yourdon.com/books/msa2e/CH15/CH15.html .
2. Jones, M. N. "HIPO for developing specifications", Datamation 22, 3 (March 1976), 112- 125. (Out of print) This citation is indirect; the original is found in:

Joshua Turner, The Structure of Modular Programs, Communications of the ACM, Volume 23 , Issue 5 (May 1980), pages 272 – 277.

3. Dan Ingalls et al, “Fabrik: A Visual Programming Environment”, Proceedings of OOPSLA (Conference on Object-Oriented Programming Systems, Languages, and Applications), September 1988.
4. Frank Ludolph et al, “The Fabrik Programming Environment”, Proceedings of the IEEE Workshop on Visual Languages, Pittsburgh, Pennsylvania, pp. 222-230, October 10-12, 1988.
5. M. Conway, “Dataflow Processing with Events”, United States Patent and Trademark Office, Patent No. US 6,272,672, filed September 6, 1995, issued August 7, 2001.
6. M. Conway, “How Do Committees Invent?” Datamation, April 1968. (Out of print)