Think About, and Build, Interactive Applications By Wiring Up Flow Components

© 2005, 2015 Melvin E. Conway

conway.mel@gmail.com

Motivation: Lessons and Design Principles

This paper elaborates on the design principles stated in http://melconway.com/humanize-the-craft.pdf, which is best read first. In addition this paper describes a concrete realization of these design principles.

Lessons. Here are some lessons that have changed the way I think about application languages and development.

1. **The internal form of the application that the computer executes should be the same as the external form that the developer sees.** What that means in practice is that if there is any translation that occurs between the external and internal forms, it must be totally undetectable to the developer. It is not enough that such a translation be very fast; any structural differences between internal and external forms will show up as confusing artifacts during the debugging process. An early example that inspired me was the programmable desk calculator, with its "learn" and "run" modes; the programming language is the same keyboard the user uses to solve his problem interactively. Mac Pascal, introduced in 1984 with the Macintosh, was a proving ground for this lesson.

2. **Any change the developer makes to the visible form of the application must be immediately reflected in correct application behavior.** The spreadsheet is an example.

3. **Application languages and algorithm languages belong to different species.** An attempt to marry them will probably produce a monster. (Example: later versions of IBM Report Program Generator (RPG).) Application languages are typically more static than algorithm languages, which is good; many people have trouble strategizing sequentially. Examples that have influenced me are: the family of IBM punched-card machines, Query By Example, and the *original* RPG. Application languages don't have to be able to build every application; they are optimized for simplicity within a specific class of applications. *The purpose of an application language is not to describe algorithms but to hide them*. This leads to lessons 4 and 5.

4. **For application languages, the *absence of explicit sequence* and an *accessible visual metaphor* are valuable.** The visual metaphor helps people who think better concretely. An excellent example is the spreadsheet, which can be thought of (and is, in fact) a tool for building a program whose function is to maintain defined relationships among a set of values.

5. **Applications can be partitioned into distinct classes**; within each class there is an underlying algorithm that is implicit at run-time and is not explicit in the language. The application language for this class is a mostly static parameterization of its underlying algorithm. A classic example is the "card cycle" of IBM tabulators; programming of these machines involves mostly data formatting, a limited set of conditions, and no sequential

algorithms. My offering is the scheduling algorithm described below in the Technology discussion in Section 7.

6. **I have settled on the flow network as the preferred visual metaphor.** Flow networks have a successful history as conceptual models of many processes. However, flow networks (I call them *wiring diagrams* here) as practical models of interactive applications have been unworkable because of intractable complexity arising from the perceived need for bidirectional flows to handle both data and events. My solution to this problem, involving only unidirectional flows, is described in this paper.

7. **My model of how the application developer works has changed from a programmer at her keyboard to a potter at her wheel**. I realized that if there is to be a breakthrough in matching the application development process to the way we are wired (this is certainly not the case today), ***we must exploit the massive investment Nature has made in the hand-eye-brain system of every human***. Just watch how a baby learns to feed herself, first struggling to pick up a raisin and put it in her mouth, later learning to use a spoon.

Six design principles. Here is how this vision of developer as potter translates into design principles for a development tool. I will use the language below of an artisan using a tool to fashion working material into an artifact. (Keep in mind that the working material and artifact are interactive applications with their own behaviors.)

1. **Unity.** There is no input ("source") or output ("object") form of the artifact. There is one working material and the development process consists of successive transformations of this working material. Corollary: During development the concept of starting an application is not fundamental; the working material is always running.

2. **Transparency.** Ideally, the tool is invisible, and the artisan has the impression of having her hands directly on the working material. In the case of software this is not going to be literally true any time soon, but design options can be weighed with this principle in mind. Well-designed word processors and spreadsheets can approach the illusion of transparency.

3. **Continuity.** Small changes to the working material produce small changes to its behavior. Of course, software can be severely nonlinear, but small changes such as adding or removing a wire or component should produce understandable effects in the behavior of the working material.

4. **Immediacy.** The effect of every change to the working material is immediately seen in its behavior. There is no build-time/run-time distinction.

5. **Interactivity.** The artisan is in an iterative conversation with the working material: a change is made to the working material, it communicates the consequences of that change to the artisan; the artisan amends the change. This is a convergent process that leads to an artifact with the desired behavior. I think of this as an unconscious dance involving the working material and the artisan's hands, eyes, and brain.

6. **Reversibility.** Any immediately recent contiguous sequence of changes can be undone.

The Technology

1 Features of the Technology

The combination of the tool and the working material is a modeless facility that enables a what-you-see-is-what is-executing-right-now style of interaction with the developer. Here are some of its features.

1. The developer assembles programs by wiring up flow components from a library in the workspace of a drawing program.

2. A multi-level abstraction and reuse mechanism permits wiring diagrams to be encapsulated to form new flow components; these are externally indistinguishable from components that are built by programming.

3. The internal structure of a running program is isomorphic to the visible structure of the developer's wiring diagram in the tool's workspace. The fact that there is no translation process between the two greatly facilitates quick response to changes.

4. The interactive wiring tool and the program under development run concurrently, not serially, with either one ready to respond to the next user-interface event.

5. Any change to the wiring diagram is immediately and correctly manifest in the program's behavior, including its output. This plus properties 3 and 4 support an interactive "what-you-see-is-what-is-executing-right-now" experimental style of development.

6. The wiring diagram can be exported to form a free-standing application.

2 The Wiring Tool's Workspace, Projectors

We shall use the simple example of Fig. 1[1] to show the handling of collections and the approach to data presentation.

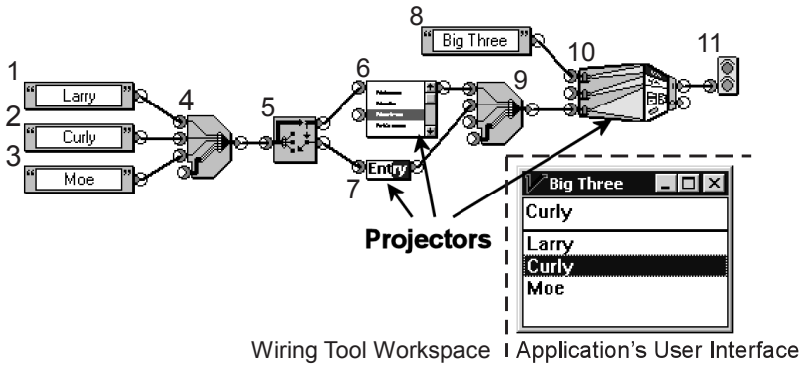

Fig. 1. Above and to the left of the dotted line is the application's "wiring diagram" under development in the wiring tool's workspace. (The dotted line, component numbers, and callouts were added to the figure after the screen shot was captured.) The lower right part of the figure shows the display 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). Typically, data source components (files and databases) are at the left and user interface components are at the right of a wiring diagram.

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 and nonfunctional.) The Selector component sends its input collection out its upper source connector for display by the List Box window-pane component 6. This List Box component is a *projector*[2], in that it "projects" the collection information into the list box window pane in the application window shown at the lower right, and receives user events from this list box window pane. The Selector component is tightly coupled to the List Box component so that a mouse event that causes the list box selection to change immediately causes "rotation" of the selector switch. (The coupling mechanism is described in Section 4.) The element of the input collection selected by the position of the selector switch comes out of the lower source connector of the Selector component and enters component 7, which projects the text line pane into the application window.

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

[2] The choice of this word is based on the metaphor that the computer display is a back-projection screen and all the program components are behind this screen.

Component 10 projects the window frame, comprising the title bar (whose text is received at the top sink connector), the menu bar (not showing in the application window because there is no input to the middle sink connector), and the rectangular data area of the window below these. 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 to Section 6 a discussion of component 11 and its connection to component 10.

3 How This Application Works

Structure of the application. The internal structure of the executing application is the same as the visible 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 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 class is instantiated at the time its icon is dragged from a component palette onto the workspace of the wiring tool. (Note, then, that a program is "running" even while it is being built, and its output is always consistent with its structure.)

A flow object is a wrapper for all application data; it provides a uniform interface between components and application data. Each flow object is instantiated by one source connector unique to it, 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 can make them visible in the workspace. *Every connector has a member that references a flow object.*

In the informal flow metaphor, application data objects flow down wires from component to component; some reach user-interface components and are projected onto the screen. This is the conceptual model used to describe the example of Fig. 1. What actually happens is somewhat different. 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. Fig. 2 illustrates a possible structure of a flow object.

If the application data object is not simple but is a structure or collection, each of its elements is wrapped by a flow object.

The references to the flow objects available to a component instance reside in the sink and source connector objects of that component.[3,4] By means of a message through one of its connectors the code specific to the class of that component communicates with the flow object referenced by that connector. A component can send a message to the owner of a flow object by means of a message chain **`getFlowObject getOwner`**[5] to one of its sink connectors.

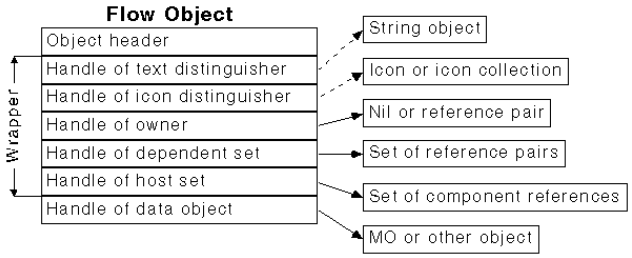

Fig. 2. Illustrative structure of a flow object.

4 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 obviates the need for explicit functional coupling of components to each other. In Fig. 1, it implicitly accounts for the tight coupling between components 5 and 6. *The update protocol is largely responsible for the fact that the application behaves as the informal flow metaphor and the program's static structure suggest.*

Fig. 3 shows the four kinds of message involved in the update protocol. Components are shown above the horizontal dotted line; flow objects are below. In Fig. 3 component A's source connector owns flow object x, references to which flow through component B to components C and D. The lower source connector of component B owns flow object y, a reference to which flows to component E.

> At component design time, reliance on the update protocol greatly enhances *decoupling of component designs*. At execution time, the update protocol implements *coupling of component behaviors* without a component's having to know what is connected to it.

[3] 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".

[4] There is a unique reference called "MO" (meaningless object) that indicates absence of a reference. It is not like nil or null in object-oriented languages because every component must behave meaningfully for MO on any sink. Newly instantiated components have MO in every sink. MO plays an important part in assuring that removal or addition of a wire or component continues correct behavior of the running program according to the wiring diagram at that moment.

[5] Message concatenation is written Smalltalk-style. Other notations write a dot where the space is.

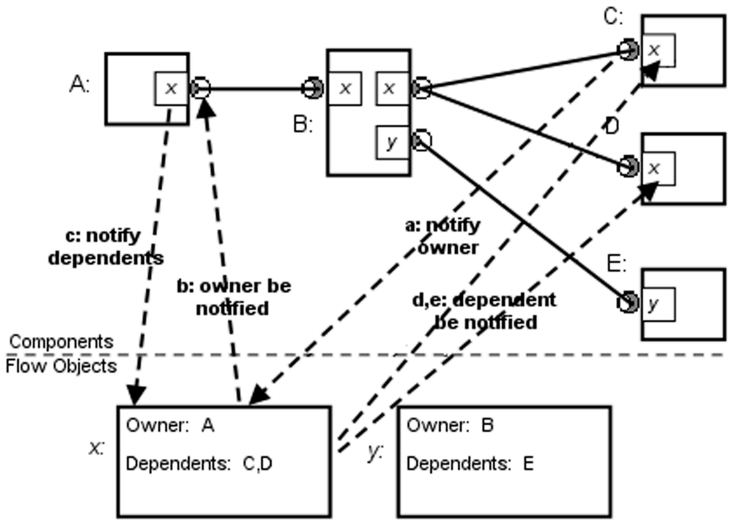

Fig. 3. The sequence of four messages that constitute the update protocol.

Such flows occur, with a few exceptions, at the time that wires are drawn in the workspace. All components receiving flows at their sink connectors perform computations defined by their class, and send the results out of their source connectors. This continues until the system is quiescent. After that no more flows typically 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 a wire is dragged from a source to a sink a flow down that wire is induced. When a wire is deleted its former sink connector is given an MO (see footnote 4) and it receives a **`dependent be notified`** message, which can induce flows from its source(s).

> When (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 (usually a projector) receives an event, alters its own state and the state of the application data referenced by the flow objects referenced by its sink connectors, and then sends a **`notify owner`** message to the flow object (via its sink connector). This event-receiving component modifies the application data in response to the event. Therefore, before step 3 below begins, the application data has already been altered to reflect the event.
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 this owner, and the flow object stores this reference. In direct response to the **`notify owner`** message the flow object 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 component-specific processing determined by its class after the source connector receives the **`owner be notified`** message. Then the owner source 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. One dependent is the component that initiated this message sequence; it remembers this and does not act in response to the **`dependent be notified`** message; the others respond to the new input.

Notice that no component needs to know what components are wired to it; this is an important aspect of decoupling within the design that contributes to component reusability.

5 Application of the Update Protocol to the Selector Component

The tight coupling between components 5 and 6 in Fig. 1 is a typical application of the update protocol. This coupling occurs implicitly without the developer's declaring it, as a direct consequence of the List Box component's upper sink declaring itself as a dependent.

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 some additional information (projected as a highlight) that tells the List Box component that Curly has been selected. The object that is being projected by the List Box component is not a list but a *selected list* derived from this list. The difference between a list and its selected list is shown in Fig. 4.

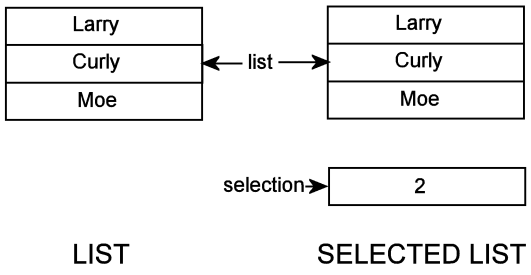

Fig. 4. The list is the input to the Selector component of Fig. 1. The selected list is what is projected by the List Box component. "Curly" is selected.

The owner of the selected list object is the upper source connector of the Selector component.

Consider what happens if Moe is clicked in the list box. First, the operating system's user-interface management system (**UIMS**) changes the visible highlight in the list box. The List Box

component 6 receives the event from the UIMS with a parameter 3, denoting Moe.[6] 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 in its selected list; it then notifies the Selector component that sent it the flow object to act on the new index value. The List Box component does this by invoking the update protocol with respect to its upper sink connector.

When its upper source connector receives the **`owner be notified`** message the Selector component can find that its index has been changed, so it changes the data object reference in the flow object owned by its *lower* source connector; it then has this connector send a **`notify dependents`** message to this flow object. Notice that there is no flow (after the first selection event). Also notice that the lower source connector remains owner of the same flow object, but that flow object now references a new data object.[7]

6 Buttons and Menus: The Command Data Object

A commonplace interpretation of the behavior of a flow component that projects a button or a menu item onto the user interface has been that, when the component projecting the button receives a mouse-click event, a right-to-left flow of this event is initiated.

There are no right-to-left flows in this program model. Instead, ***handling of an event at a button or menu item follows directly from the update protocol***. This occurs through the agency of an application data object called a ***command***. *A user-interface component that shows 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. 5 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 "File" menu is opened in the application window. The actual behavior, described next, is not so different.

The Beep component has a specific 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 of these owned by the source connector. When the command flow object makes its way through the Add Label component it acquires the label[8] "Ding". This label overrides any default label in the command flow object. A collection containing (in this case) only this flow object enters the "File" Menu component. Upon receiving this collection of command flow objects the "File" Menu component declares itself to be the dependent of every command flow object in the collection. (In Fig. 5 the one

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

[7] The reader may wish to satisfy herself that if *two* list boxes A and B are wired to the upper source connector of the Selector component and A is clicked, then B will track the change.

[8] The Text Distinguisher in Fig. 2.

menu item in the projected menu is the projection of the beep command, with the text "Ding".) The "File" Menu component's source connector then sources its own command flow object to the center sink connector of the Window Frame component, which works with the resident UIMS to project an opened menu when the menu's title "File" is clicked.[9]

When the mouse is released on the "Ding" menu item the Window Frame component is notified by the UIMS with a parameter indicating that the "File" menu was the one chosen. This causes the "File" Menu component's source connector to receive an **`owner be notified`** message with a parameter denoting the "Ding" menu item. The "File" 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 flow object it is referencing; this is relayed to its command data object. (This sequence of steps, from the mouse event to the **`invoke`** message has elsewhere been called "picking" the command.) 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 "File" Menu component (because it has declared itself to be the flow object's dependent) the "File" 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.[10,11]

A command data object has its own state, in particular, an enabled/disabled flag. This state can be projected onto buttons and menu items by their enabling/disabling; in Fig. 6 the Undo menu item is disabled.

We are now ready to 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 On Go component, ultimately causes the window to open.

---

[9] Typically the input to the center sink connector is a collection of collections of command flow objects.

[10] All these behaviors, from the mouse event to the menu closure (except for the particular behavior of the command data object), are inherited by the respective components.

[11] The reader may have wondered about the purpose of the lower sink connector of component 6 in Fig. 1. It picks an incoming command when the selected item in the list box is double-clicked.

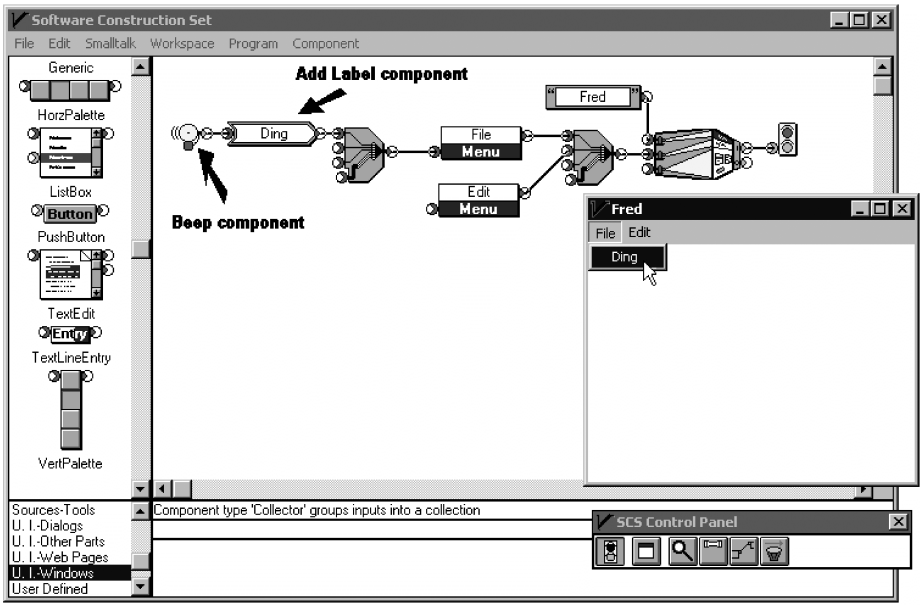

Fig. 5. The program beeps when the "Ding" menu item receives a mouse click. The wiring diagram and the application window are both shown in the context of the wiring tool.

In Fig. 5 notice the little floating window at the lower right labeled "SCS Control Panel". The leftmost button in this panel is the stop-start button.[12] The icon on the button is a traffic light that toggles between red and green. Clicking this button when it is red causes component 11 of Fig. 1 to pick the command flow object at its input, ultimately causing the window to open. The stop-start button keeps a list of all the On Go components in the application. After signaling them all it changes its traffic-light icon to green. (One might imagine an On Stop component that picks its input when the button flips from green to red. Otherwise, there is no concept of starting or stopping a program. A component is running if it is in the workspace of the wiring tool.[13])

[12] The other buttons are for debugging, message tracing, and single stepping.

[13] For some extreme development situations there is a provision to reset and restart all components of a program. When a wiring diagram is exported from the wiring tool to become a freestanding application, this is how the application is started.

7 The Scheduling Algorithm

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

An important scheduling consideration is that a multi-sink components does not know how many of its inputs will change and/or in what order, and it is uneconomic for such a component to recompute its outputs immediately in response to every flow or **`dependent be notified`** it receives.

My approach to both of these issues has been to defer processing of any change message arriving at a sink connector; rather, the component is put into a pending state and a callback is added to the end of a queue, provided it is not already there. (This is necessary only for components with more than one sink connector.) Control returns to the queue manager, which de-queues the first-in pending component for resumption of processing. When the queue is empty the processing of an event is complete.[14]

8 Abstraction

It is possible to encapsulate a wiring diagram, turn it into a component, and place the component into the library, which is projected onto one or more component palettes. For this to be useful we need the flow equivalent of formal parameters. This function is provided by *Connector components*.

We will now cause the wiring diagram of Fig. 5 to evolve into a simple reusable text editor component by adding a Text Edit component from the library as shown at the bottom of the wiring diagram in Fig. 6.

[14] There are three queues in the prototype that are emptied in sequence. The last queue to be emptied holds callbacks that repaint windows; this tactic minimizing flashing of the display.

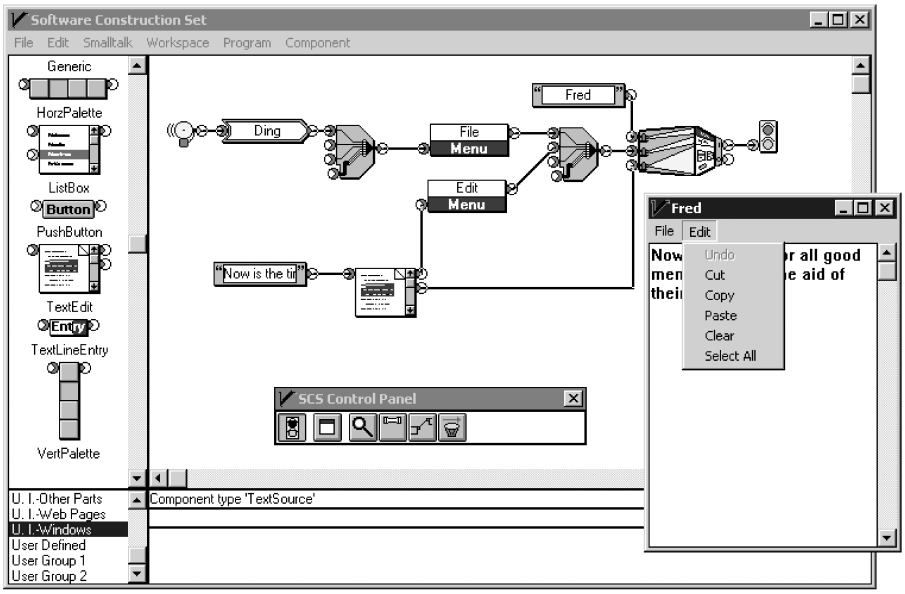

Fig. 6. An edit component from the library is added to the beep program.

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

Now let us remove the beep logic and prepare to encapsulate the remaining 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. 7. (The vertical crosshatch in the Connector component's icon is meant to suggest the wall 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)references to flow objects are the only things that pass through sink and source connectors no matter how many levels deep their components are, (2)all flows are in the same direction, and (3)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 wiring tool even those components that are presented to the developer as part of a basic set might have inside them not code but wiring diagrams.

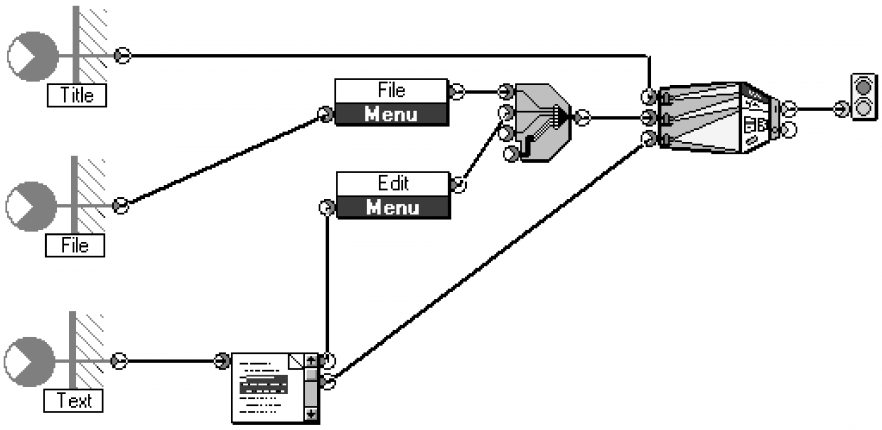

Fig. 7. Three Sink Connector components are added in preparation for encapsulation.

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, create a modified derivative of it.[15]

9 The Wiring Tool and the Application Under Development as Peer Programs, or 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 the text line pane, edit "Moe" by removing the last letter so it becomes "Mo". Fig. 8 shows the result.

Notice that "Moe" has become "Mo" not only in the text line 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. (Note the extension of the *projector* concept to the wiring tool.) This internal structure happens to be the program under development 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 correct program behavior.

[15] The permissions granted to the wiring tool by a component depend on how the component is licensed. There is a discussion in the patent [1] of a technological response to this set of issues.

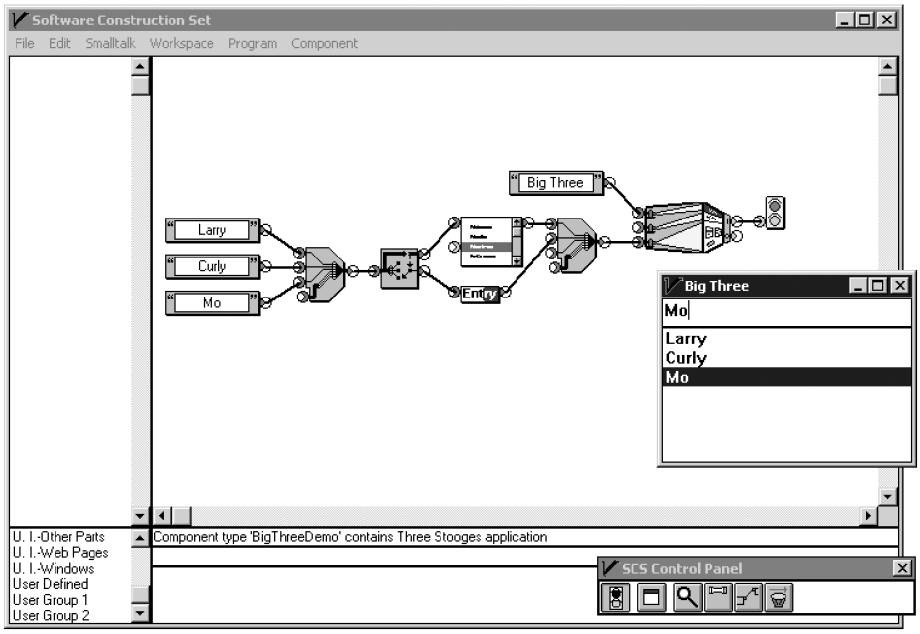

Fig. 8. The application of Fig. 1 in the wiring tool's workspace after selecting "Moe" and editing the text line by removing the last letter.

Thus, the program may be modified on the fly and will continue to behave according to its wiring diagram *at that moment*. The user's experience is similar to operating a contemporary word processor: what you see is what you get, without delay. This immediate what-you-see-is-what-you-get behavior encourages an experimental style of development because the developer immediately sees the result of every change. Wiring can be built incrementally from the inside (left end) out and/or from the outside (right end) in. This incremental style is further enhanced if real application data are present and if temporary inspector components are wired into strategic places.

The sequence of Fig. 5 to Fig. 6 to Fig. 7 illustrates the use of a placeholder or scaffolding in experimental development.

Because either the wiring tool or the application under development can accept the next user-interface event, depending on the position of the mouse, there is no logical distinction between the wiring tool and the application under development. The wiring tool has projectors that respond to events in the same way as the application under development, and they can both be considered parts of the same program.

10 Web Applications

The freestanding-PC style of the applications shown in this paper reflects its twentieth-century history. It is a short step to applying the wiring model to distributed applications. I have built a rudimentary restaurant order-entry demo in which the projectors (in particular, tiled-button projectors of collections) were implemented in JavaScript running in a browser on a WiFi-connected tablet, with the other components collocated with the server. Ajax conventions were used for message passing between client and server components.

11 Reuse

The update protocol is an effective decoupling mechanism, so the principal sources of coupling (and therefore the principal determinants of reuse) are the public message sets of the application data objects in use.

Two attributes of this program model, unidirectional flow and the uniformity of all flowing entities (because every application data object is wrapped by a flow object), greatly enhance reusability. Other wiring models without these attributes, for example those that distinguish between data flows and message flows or those with bidirectional flows, restrict reusability.

Following is a vision 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 more stable 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 of these applications will have long lives and will evolve, and some will be built for one use.

The architecture similarly [2] contains within it a basis for dividing responsibility in the enterprise between two classes of developer. 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. On the other hand, it is conceivable, perhaps even reasonable, that the libraries of manufactured wired 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. I envision a uniform *message selection interface* presented to 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. Provided that wiring is done with real data present, the application builder can query the current set of public messages of a business object, possibly with a pop-up menu, and select from this set. Each public message will come with a help facility and a template for parameters. There will be a set of messages inherited by all business objects to support this uniform message selection interface, which will largely be implemented in the wiring tool.

References

1. 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.
2. M. Conway, “How Do Committees Invent?” Datamation, April 1968. (Out of print. The thesis of this paper has come to be known as “Conway’s law.”)