The *Flow Object* Pattern

This paper specifies the Flow Object Pattern. which defines the communication structure of applications that are written in a wired-component language using a wiring tool that complies with all the attributes of a modeless workflow. The Flow Object Pattern provides components with access to application data visible to these components, it decouples components from each other and from all other application data, and it provides a uniform mechanism for synchronization of changed application data.

Twitter: @conways_law

July 22, 2017
conway.mel@gmail.com

Motivation and an Example

Source-Object Isomorphism

If

you build an application using the *wiring model* (there is an example below)

and

the wiring tool you are using supports all the attributes of a *modeless workflow*[1]

then

the structure of the underlying application execution machine will necessarily be isomorphic to the wiring diagram

and

this machine will be built on the Flow Object pattern.

This looks like a rule, but it's really just an observation based on years of playing with these ideas. I have found no other way to realize fully a modeless workflow. Given that, the question arises: how does an application structured this way work? The Flow Object pattern contains the answer.

This paper describes the Flow Object pattern. It is closer to a specification than an earlier paper that describes the operation of the execution machine[2].

An Example of a Wired Application

The figure below is a screen shot from the working wiring-tool prototype, with component numbers added to support this description. The screen shot was taken shortly after clicking "Moe" in the list box.

Both the left-to-right flow diagram (the "source language" in conventional terminology) and the user interface of the running artifact program are shown side-by-side just as on the display of the tool, with the artifact's UI at the right.

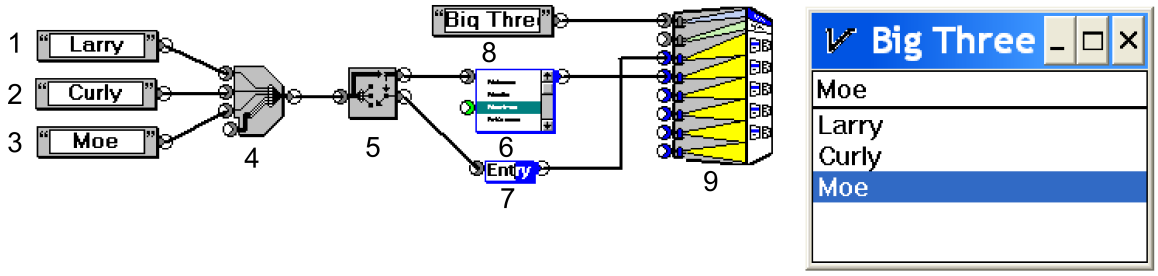
In accordance with the "Symmetrical" attribute #2 of the modeless workflow attributes, the wiring tool and the artifact are peers, and the builder can click on the UI of either one at any time.

---

1 http://melconway.com/Home/pdf/modeless.pdf page 8

2 http://melconway.com/Working/WP_9.pdf

Twitter: @conways_law
July 22, 2017
conway.mel@gmail.com



Components 6, 7, and 9 are called "projectors"; they have the particular responsibilities of rendering their inputs on the user interface of the artifact. Component 9 projects the window frame, including the title bar (and, if present, a menu bar), component 6 projects, in the form of a list box, the collection appearing at the sink (input) connector at its upper left, and component 7 projects, in the form of a text line, the selected string appearing at its sink, in this case, "Moe". This selected string "Moe" is the output (at the lower source connector) of the "Selector" component 5, which acts like a rotary switch, implicitly and tightly coupled[3] to the projected list box. The upper source connector of the Selector sources the collection ["Larry","Curly","Moe"], and the lower source connector sources the item selected, in this case "Moe", in the list box.

Component 4 is a "Collector" component, which aggregates its inputs to create an indexed collection[4]. Component 4 in particular is sinking three individual string-valued objects and is sourcing the collection ["Larry","Curly","Moe"]. Components 1, 2, and 3 are "Text Source" components that source the literal constants attached to the components by the builder.

[3] The implicit coupling mechanism is the Update Protocol, discussed below.

[4] "Indexed Collection" is Smalltalk-speak for a linear structure accessed by an integer value, frequently called an Array elsewhere.

Twitter: @conways_law
July 22, 2017
conway.mel@gmail.com

The Application's Conceptual Model

The implicit operation of the application conceptual model is *left-to-right flow of objects along the wires, from source to sink*. This conceptual model is faithfully realized by the tool and the artifact according to all the modeless workflow attributes[5], but in fact under the wraps there isn't that much "flowing" going on during the operation of the artifact.

1. In many cases, the Update Protocol in particular, there is no flow at all but a sequence of messages.
2. The application data objects are stationary and never move along the wires; when a flow does occur (expressed in the component code as a "push" message to the source connector), a reference is copied from the source to all connected sinks.
3. Where flowing throughout the wiring diagram does occur is at the time the application is started. Then the source and sink connectors are populated with object references. This initialization establishes the initial connections between components and data.

The object references in source and sink connectors are not references to application data objects but references to *wrappers of application data objects* called *flow objects*.

> *How flow objects mediate the access and modification of application data by components in a way that effectively decouples these components is an important consequence of the Flow Object pattern.*

There are two aspects to this conceptual model that enable the design goal of practical multi-level reuse of components built by encapsulation of wiring diagrams.

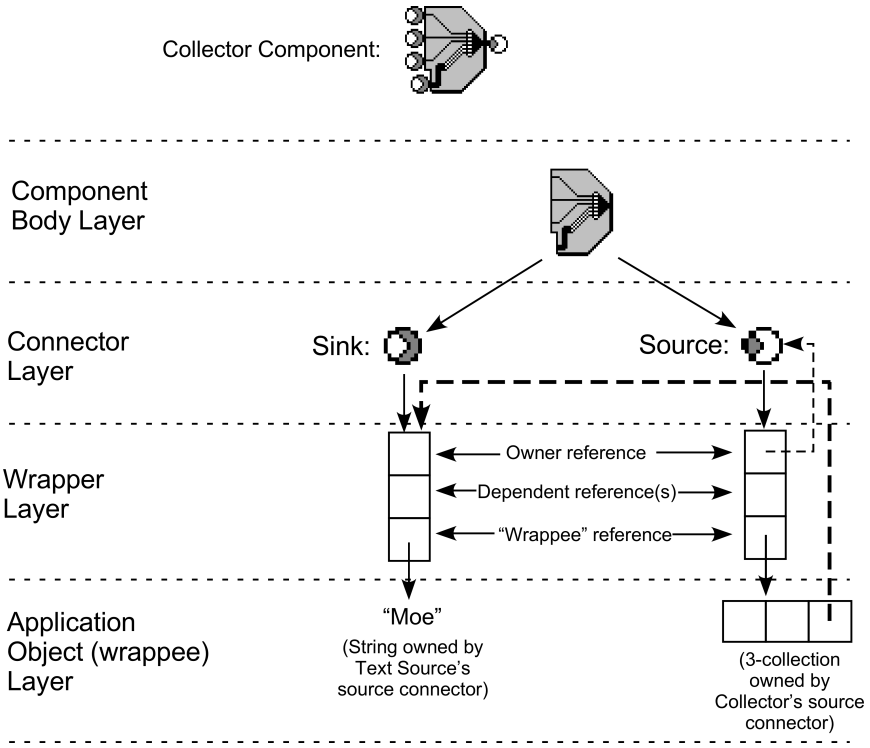
1. All components (via their connectors) see a single interface to all application data objects, presented by flow objects.
2. All flows are unidirectional, determined by the two connector types, source and sink.

---

[5] Ironically, the one attribute this prototype doesn't implement is the most common one: Undo.

There Are Four Layers In This Pattern

We will describe these four layers using Collector component 4 as a concrete example. Here is a partial schematic drawing of the component and its four layers. Only representative objects, but not all objects, are show in the bottom three layers.



Function: The function of Collector is to create and source an indexed collection whose elements are derived from the component's sinks. The first three elements come from the top three sinks (if all inputs are present). The fourth sink permits daisy-chaining Collectors to create larger collections; it accepts an indexed collection that is concatenated to the collection created from the first three sinks.

Visibility: Each connector has an association to zero or one application objects. The component body code can directly address all the connectors attached to the component. However, the component body code cannot see the objects associated with these connectors (i.e., the strings "Larry" "Curly" and "Moe" and the output triplet) until it obtains references to them; then it can send messages to these application objects. It obtains these references from the respective connectors. Specifically, sending the message `target` to a connector returns a reference to that connector's application object.

Indirection through Wrappers: Wrappers provide a layer of indirection between connectors and their application objects. This indirection provides a uniform interface between components and application objects. Wrappers also manage synchronization of application objects and their projections after receipt of an event. This synchronization is inherited run-time behavior that is not managed by the component body code. It is discussed later in this paper under "The Update Protocol".

Wrapper structure: Every wrapper has three parts.

- **Owner.** This is the reference to the source connector that created this wrapper; this is its owner. Every wrapper has exactly one owner.
- **Dependents.** This is a list of zero or more references to sink connectors. Dependents are the sinks that must be notified if their application objects change.
- **Wrappee.** This is the reference to the application object.

Every wrapper wraps one application object, and each application object referenced by a component is wrapped by one wrapper.

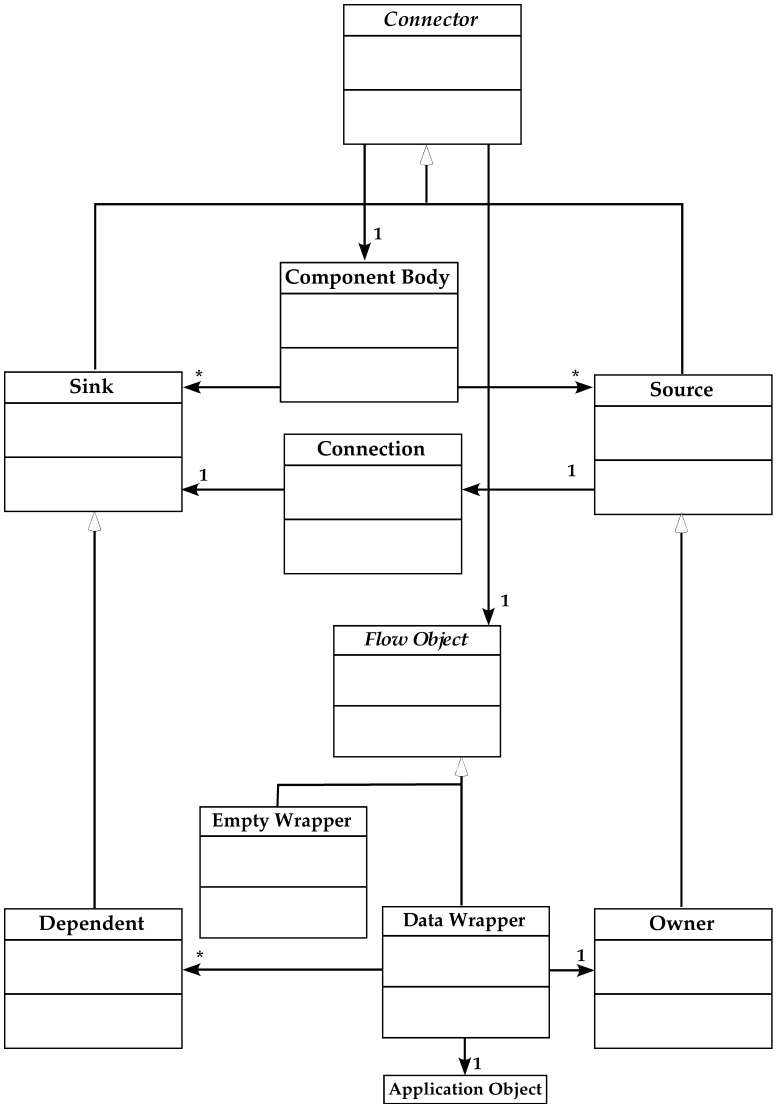
If an application object is a collection or other structure, the *elements* of the structure, rather than being other application objects, will be wrappers wrapping these application objects. This insures that every application object, no matter how deeply nested, is wrapped and can be synchronized.

In this example the source of the Collector component indirectly references a 3-collection ["Larry", "Curly", "Moe"]. The heavy dotted arrow in the figure shows that the third element of the collection references, not the string "Moe" but the wrapper of "Moe". The light dotted arrow shows that the source connector is the creator and owner of this 3-collection.

Description of the Pattern

Class Structure

The class diagram of the Flow Object pattern is shown here.



A component body (there are 9 of them in the above example) can have any number of sink connectors (conventionally shown at the component's left) and/or source connectors (conventionally shown at the right). All the processing of application objects is embodied in component bodies, implemented by sending messages to application objects indirectly through connectors and data wrappers. Each component body can directly address only its connectors but does not directly address flow objects.

Every connector knows about two objects: its component body, and some flow object. In addition, each source knows about a connection for each attached wire. Each connector that references Application Data (i.e., that does not reference the sole Empty Wrapper[6]) must respond to a message (let us call the message `target`) that answers the application object indirectly referenced by this connector.

Each data wrapper references one application object. Each data wrapper must respond to a message (let us call the message `yourData`) that answers its application object. The operation of the `target` message sent to a connector, then, is to answer the application object that is the result of sending the `yourData` message to the data wrapper the connector references. (Sending `target` to the sole Empty Wrapper returns *nil* or something similar, depending on the implementation.)

Owners and dependents exist to implement the Update Protocol. Some sources are owners; some sinks are dependents; these properties are given to connectors at their creation according to the roles the connectors play in their components. The role of a dependent is to guarantee that it will be notified if its application object changes so that its component can in turn be notified and, if it chooses, reprocess its inputs.

Every data wrapper has exactly one owner and can access it; the reference to its owner is created when the data wrapper is created.

Every data wrapper has zero or more dependents. A data wrapper first learns of a dependent the first time (a reference to) it is copied to a sink that knows itself to be a dependent; at that time the sink adds itself to the data wrapper's dependents collection. (Also at that time the sink must remove itself from the dependents collection of the data wrapper whose reference, if it exists, is about to be overwritten.)

> If an application object is a collection or structure created by a component such as Collector, and an element of the structure is referenced by a data wrapper (this can be the case with component 7 in the example) that

[6] The Empty Wrapper denotes the absence of a wire on the referencing connector or propagation of an empty input into a component. When a component is first dragged onto the tool's workspace all its sink connectors refer to Empty Wrappers.

element must be wrapped. The Collector insures this by sourcing and owning a new structure whose elements reference the component's input wrappers.[7]

Because the number of levels of dereferencing to reach an application object is unknown *a priori*, indirection caused by wrapping of application objects must be transparent. This is insured as follows. Every operation of the code in a component body that must be assured that its operand is indeed an application object does this, not by operating directly on it, but by sending `yourData` to it and operating on the result. In turn, the response of a data wrapper to the `yourData` message is not to return its application object but to return the result of sending the `yourData` message to its application object. This does not result in an infinite loop because *every object except a flow object returns itself* in response to the `yourData` message; data wrapper returns the result of sending `yourData` to its application object, and Empty Wrapper returns *nil*.

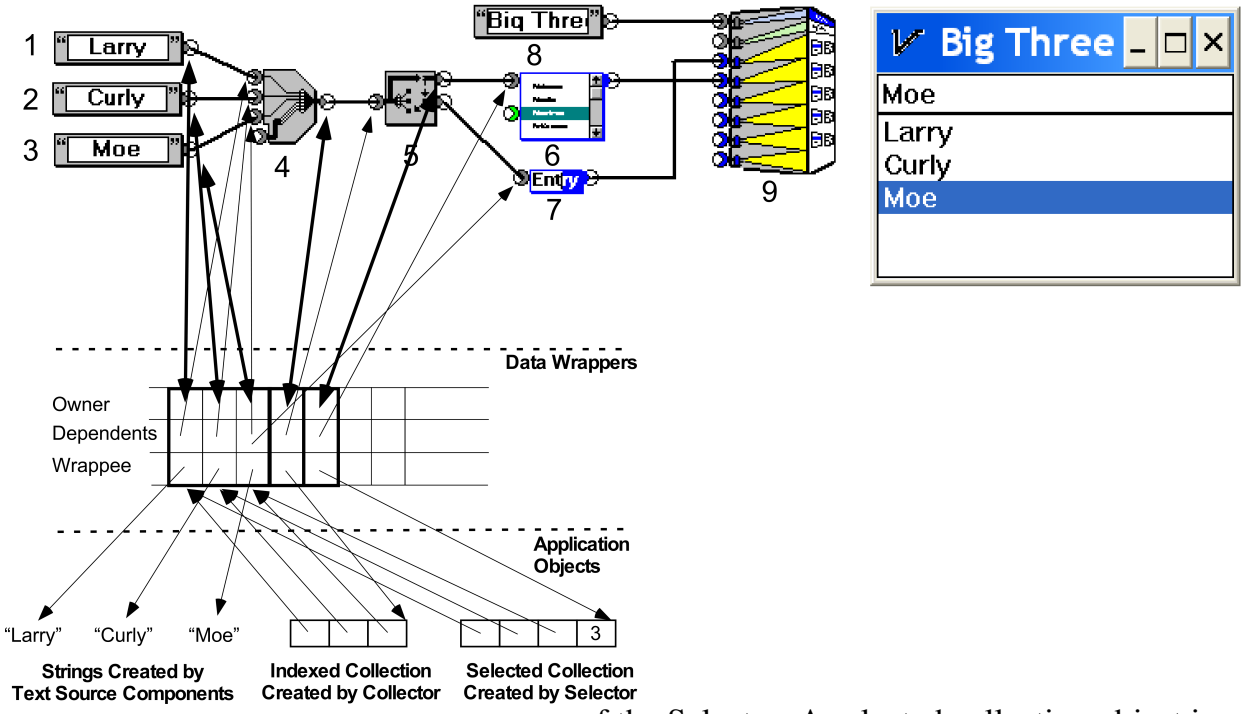
Schematic Diagram of the Example

The figure below shows the object references in the example. (References from connectors to owned wrappers are shown as heavy double arrows. Other references from connectors that are not owners are not shown.) When a source first creates a wrapper it makes itself the owner of that wrapper.

There are five data wrappers; we can consider them in three groups, bordered by heavier boxes.

1. The left three wrappers wrap the three strings.
2. The fourth wrapper wraps an indexed collection object created by the Collector component. The wrapper is owned by the source connector. The elements of the indexed collection are references to the *wrappers* at the three inputs of the Collector.
3. The fifth wrapper wraps a *selected collection* object created by the Selector component. The wrapper is owned by the top source connector

[7] There are scaling issues here that might require lazy evaluation approaches in a real system. These issues are not addressed in the prototype.



of the Selector. A selected collection object is like an indexed collection object but with an additional instance variable. This new instance variable carries the index of the selected element, which is projected by the list box as a contrasting color in the selected line of the displayed list.[8] A change to the selection (caused by clicking on a new line in the list box) is a change to this object; this change triggers the Update Protocol.

Note that the lower source of the Selector component is not an owner. If the selection in the list box changes, the Selector component pushes the newly selected *wrapped* element of its input collection out this source. This wrapper contains the reference to its owner, in this case the source connector of the Text Source component. This flow-through without ownership is what makes the use case below work.

[8] In this Smalltalk implementation indexed collections are 1-based.

Messages in the Pattern

The `push` Message

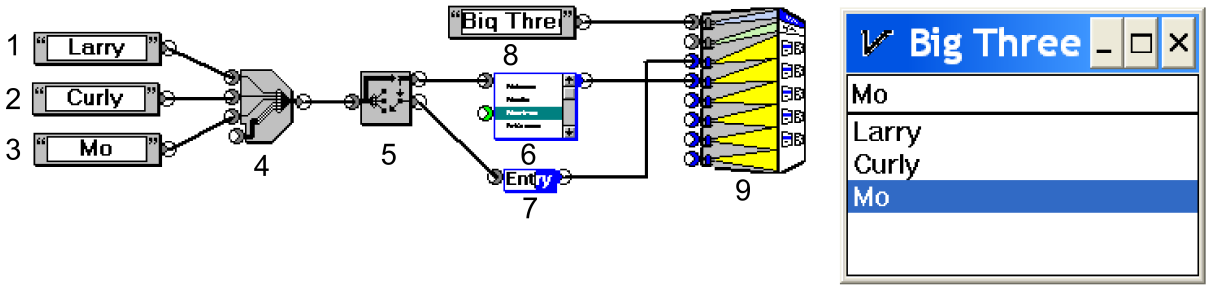
The effect of sending `push` to a source is to copy the source's Wrapper reference to every connected sink, and to cause the sink's component to recomputed using the new input.

When a source receives a `push` it sends a `transmit` message to every associated connection, with the source's Wrapper reference as an argument. The connection then sends a `receive` message to its associated sink, with the same Wrapper reference as an argument. The sink then assigns the Wrapper reference to itself; if the sink is a dependent it must first remove itself from the dependent list of the Wrapper whose reference is about to be overwritten and then add itself to the dependent list of the new Wrapper. Then the sink sends a `compute` message to its component body.

Use Case: Edit a String

This use case is interesting because it shows that there is only one copy of the string "Moe" in the combined wiring tool/artifact system. In this example the projection by component 7 of the string "Moe" will be edited.

Component 7 permits editing. In the case of this component, the user confirms the edit by striking the Enter key. The figure below shows the display after the string "Moe" in the top text line of the artifact's UI is edited by removing the final "e" and striking the Enter key.



If you consider the tool and the artifact to be one program (they are[9]) you see that the *single* underlying string "Mo" has *three* projections that must remain synchronized:

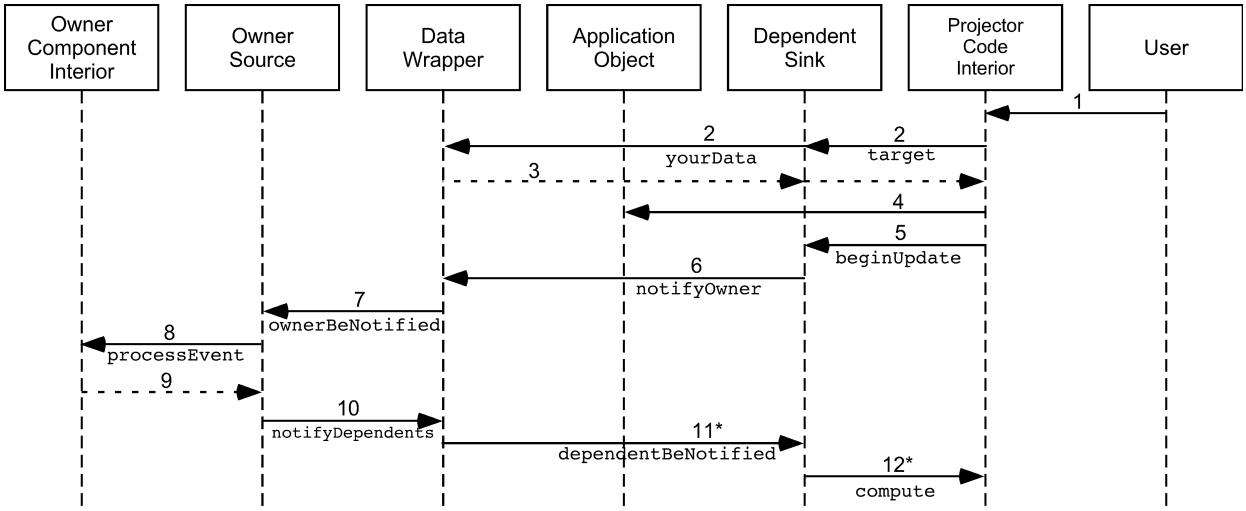
1. The text line projection (in the artifact's UI) created by component 7,
2. The list-box projection (in the artifact's UI) created by component 6, and
3. The text projection inside the component icon (in the *tool's* UI) created by component 3.[10]

[9] The design provides for separating the artifact into a free-standing application. The separation is cleaner than in Smalltalk. See http://melconway.com/Working/WP_9.pdf

[10] In the current prototype the tool is built in Smalltalk. Bootstrapping the tool using itself is a longer-term goal. When this happens the wiring workspace of the tool's user interface will be a projection of a wiring-diagram object. Every Component in the wiring diagram will be responsible for projecting its own icon, and a connection will project as a wire.

The Update Protocol

The Update Protocol is a sequence of messages that assures synchronization of an application object and all its projections.[11] It comprises messages 6 through 12 of the following sequence diagram.



In this example the user edits the text projected by component 7. Usually a User Interface Management System (UIMS) does this in a buffer normally hidden to the projector code. The UIMS immediately renders changes the user makes to the text. Then the user submits an event (the Enter key), which causes projector component 7 to obtain the current text value from the UIMS and trigger the Update Protocol, propagating notice of the modified data to all dependents.

1. This is the user interacting with the text via the UIMS.
2. Steps 2 and 3 comprise the function of the `target` and `yourData` messages. The projector code sends the `target` message to its sink connector, which sends the `yourData` message to its data wrapper.
3. The `yourData` message returns to the sink connector a reference to the application object[12], and the `target` message returns that to the projector code.

---

[11] Viewing the operation of the Update Protocol more generally, it brings the wiring diagram viewed as a constraint network back into equilibrium after an event.

[12] In a pure object-oriented language such as Smalltalk it is said that "everything is an object". In truth, everything *that the code touches* is *a reference to an object*, so "a reference to" is dropped as

4. When the string projector component 7 receives the confirming Enter event from the UIMS, it obtains the current value of the string from the UIMS and sends a message to the application object, setting it to this value, using the reference obtained in step 3.
5. The projector sends the `beginUpdate` message to the sink, which starts the Update Protocol.
6. The sink sends a `notifyOwner` message to the data wrapper. (Before doing that the dependent sink sets a flag that will block double rendering; see step 12.)
7. The data wrapper sends an `ownerBeNotified` message to the owner.
8. The owner sends a `processEvent` message to its component's code interior. At this point the component's code interior has the option of reacting to the user event. This normally occurs only if the application object being projected is a Do-It[13]. If it is a Do-It, the user event is a pick of the Do-It's projection and step 8 causes the interior of the owner component to perform the pick action.
9. This is the return from the `processEvent` message.
10. The owner sends a `notifyDependents` message to the data wrapper.
11. The data wrapper sends a `dependentBeNotified` message to *each* of its dependents.
12. For each dependent: if the flag is not set the dependent sends a `compute` message to its component's interior, which accesses the data

---

endlessly redundant. More to the point, the subject of programmer documentation is function, not implementation, and dererencing does not participate in the function, only the implementation. (This is a source of difficulty to beginners because the same object can be in many places at the same time.) In pure OO-speak this sentence would say that the `target` message *returns the application object*.

[13] A Do-It is an application object whose projection is a clickable object. Clicking on this projection then causes some "pick action" to occur. In this manner the introduction of the Do-It type in connection with the Update Protocol obviates the need for right-to-left flows for handling user events like button and menu clicks.

via the dependent (see steps 2-3) and refreshes its projection if it is a projector. If the flag is set all that happens is that it is reset.

This description accounts for component 7 and similar projectors to be updated. However, other updates might be necessary, for example the line in the list box projected by component 6. Here is how that happens.

The data wrapper for the string "Mo" has *two* dependents; the other one is the third sink of Collector component 4. In step 12 above this sink sends a `compute` message to the Collector component body. This causes it to recompute its output.[14,15] This causes the source of component 4 to push its new output to component 5. component 5 then recomputes, and the push then goes to component 6, which recomputes and re-renders the list box.[16]

---

[14] This causes another flurry of messages, and eventually the network settles down in its new state. This is discussed in more detail in "Component Computation Scheduling, pages 13-14 of http://melconway.com/Working/WP_9.pdf . In developing this scheme I was inspired by Alan Borning's iterative constraint-solution work with ThingLab https://github.com/cdglabs/thinglab .

[15] There are optimization opportunities here. The response to *changing the application object value of one input* to a Collector should not be so drastic as the response to *adding another input* to a Collector (when a wire is added). In this former case, changing the Collector input does not need to produce a push, but only a partial Update Protocol beginning with step 10 from its source. The Collector is able to make this distinction.

[16] In this use case (in which the input collection itself is unchanged and the selection is not changed) there is no change at the second Source Conncetor of Component 5. (Because of the different types of change that can occur the Selector is one of the most difficult components I've encountered.)

What Components Do

Almost all component types are pure functions. That is, they compute values based on their inputs and they source these values directly.

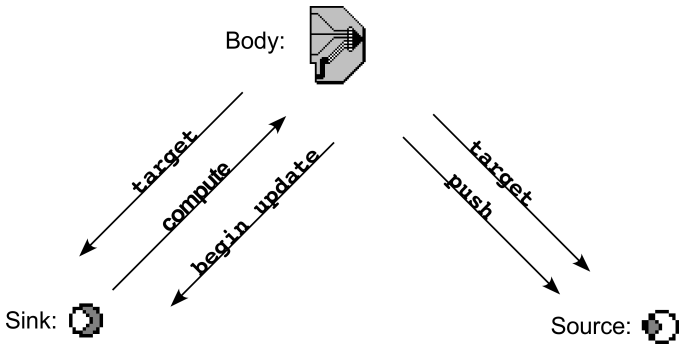
The contract that specifies the output(s) of a function as determined by its input(s) is often called its transfer function. I'll use that term here.

Components fall into two broad groups.

1. **Primitive components:** The component body is built from code. This code defines the transfer function of the component. *All of the preceding description is about primitive components*.
2. **Composite components:** The component as it appears in the tool's wiring workspace is an encapsulation of a wiring diagram that was built in a wiring tool, encapsulated, and added to a library. Then the component was pulled from the library into the workspace.

What Primitive Components Do

There is a small, well-defined message protocol between the bodies of primitive components and the connectors seen by these components. This message protocol is the interface that the code of primitive component bodies must support. It is shown in this figure.



What Composite Components Do

Composite components are convenience creations that effectively implement abstraction and multiple-level component reuse.[17] Appearing in the wiring tool's

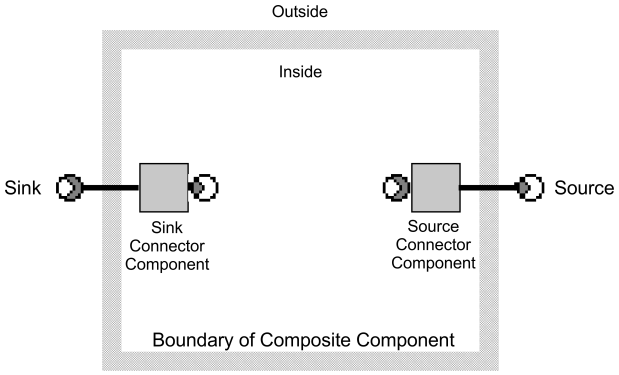
[17] Notice the parallel to Smalltalk methods; primitive components are analogous to Smalltalk-80 primitive methods, which are implemented in code. Just as Smalltalk-80 is based on a set of primitives whose size is $O(2^7)$, my goal is that the wiring model will be analogous. The effort to bootstrap the tool will be the test of this goal.

workspace they act as shorthand proxies for wiring diagrams. Their behaviors are described here.

We can think of a composite component as having an inside and an outside. The inside of a composite component is a wiring diagram. The outside of a composite component is a set of (zero or more) sources and (zero or more) sinks that follow the same wiring rules in the tool's workspace that sources and sinks of primitive components do.

Connectors on the outside of composite components do not send or receive messages, they cannot be owners or dependents, and they do not participate in the Update Protocol. What they do is re-route connections.

Each connector on the outside of a composite component is paired one-to-one with a special *connector component* on the composite component's inside. Each sink connector is paired with its own *sink connector component*. Each source connector is paired with its own *source connector component*. The relationship between a connector and its connector component can be depicted as follows. (The heavy lines between the connectors on the outside and the connector components on the inside are not wires; they are just graphic elements showing a relationship.)



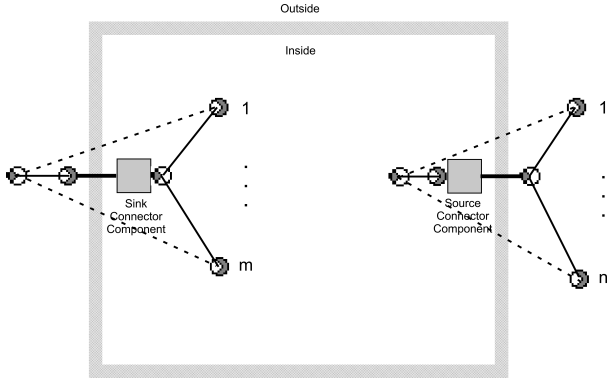
Notice that the sink connector component has a *source connector that can be wired* in the interior wiring diagram, and the source connector component has a *sink connector that can be wired* in the interior wiring diagram. How these internal connectors are wired defines the connections between the inside and the outside of the composite component.

Connector Functional Specification:[18] Sink and source connectors on the outside of a composite component together with their corresponding internal connector components do not participate in flows; they define wiring connections between primitive-component connectors in the inside and primitive-component connectors on the outside of the composite component. This will be described now.

First, let's review the wiring rules.

- A connection connects one source connector and one sink connector.
- A sink connector can have zero or one connection.
- A source connector can have any number of connections, including zero.

The following figure shows how the connections between outside connectors and inside connectors are defined. The solid wires are what the person building the application sees in the wiring tool. The dotted wires are the connections that the execution machine sees and executes. Of course if no wire is connected to the outside connector then there is no rerouting.[19]



[18] When a wiring diagram is encapsulated and put into a library it is given outside connectors in exact correspondence with the connector components in the wiring diagram. After the library component is added to a wiring workspace this rerouting occurs when a wire is added to or removed from an outside connector. Keep in mind that, except for these changes, connections inside a composite component are immutable.

[19] This functional specification can be directly implemented. This implementation has the performance advantage that the time it takes a sink to execute a `receive` is independent of the number of "walls" (layers of abstraction) through which the flow goes. There is a simpler implementation that does not have this advantage in which the sink executes the `receive` by sending a `push` to the source on the other side of the wall.