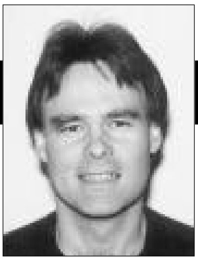
Dwight Deugo

Wayne Beaton

Managing Connection Complexity

You're given your first visual programming assignment. You spend the day working on it and are proud that you completed it without writing one line of Smalltalk. A week later, after being asked to add a new feature, you return to your picture (the visual equivalent of code) and realize that you can't remember the semantics of all of those connections. It takes the rest of the day to understand what you developed last week and add the new connections to support the additional feature. Does this sound familiar?

Often we are asked "Even though visual programming enables developers to create windows quickly, what approach should one use in order to minimize the connection complexity of visual parts? "The approach should enable developers to quickly and easily understand what they're building now and what they've built in the past.

One source of the problem developers are experiencing is a result of working in a new paradigm (the construction from parts paradigm) with little or no training. When the jump was made from assembler to structured programming, many developers wrote spaghetti code until they were educated in structured programming techniques. Now, for the same reason, many developers are painting spaghetti visual parts. The one difference today is that you can see the mess you've created for yourself. However, with a little care, this need not be so. Visual programming environments, such as IBM's VisualAge, provide an assortment of parts and facilities for decreasing the complexity of visually programmed systems.

In this column, we examine one technique for managing the connection complexity of visual parts. This technique is called factoring. Too often we see a window, a Client Profile Editor for example, containing every connection to support editing of a client's name, address, phone numbers, credit history, and more, only to see other windows provide the same support for viewing or editing the identical information. One problem is that the Client Profile Editor has too many connections to be understandable, especially when you add those to support menu interction. Another problem is that the same connections are used in every window that supports the editing of the clients address.

The simple and easy solution to these problems is to encapsulate the connections and parts that support the editing and viewing of a business object into one reusable part. The composite part can be placed in any visual part, and connect to its required business and supporting objects. One can significantly decrease the number connections in any visual part that uses reusable parts, because the reusable part manages its own connections, and they are not visible to surrounding visual parts. Also, changes to the reusable part's behavior or visual appearance are immediately reflected in any part that uses it, avoiding any potential maintenance problems that might occur when having the same functionality implemented in two or more places.

Here, we make use of two reusable forms to minimize the connection complexity of a client profile editor. The window supports editing of a client profile, including a client's name, age, and address. The reusable forms are views of two business objects: a ClientProfile and a CanadianAddress. Like the editor, they support the cancellation of edit changes.

THE CLIENT PROFILE EDITOR APPLICATION

The Client Profile Editor lets one edit and, if desired, cancel any changes to an existing ClientProfile. For this example, we do not show how the ClientProfile is loaded or saved. Our goals are to minimize the number of visual connections needed to meet the editor's requirements, and to permit Smalltalk code only when the operation can't be done visually. The rule-of-thumb, "less is better," is true when it comes to visual connections. Fewer connections make a window's implementation easier to understand, and easier to maintain.

A ClientProfile contains a CanadianAddress, and it is from this object that we begin our exercise. Although our requirements have the address displayed from only a Client Profile Editor, we decided to create a reusable CanadianAddressForm to display it. We realize that addresses are often modified, or simply displayed in many different windows, and we want all future windows to display them in the same format. Encapsulating the logic for viewing

Dwight Deugo and Wayne Beaton are senior members of the development and educational staff at The Object People, in Ottawa, Ontario, Canada. Dwight (dwight@objectpeople.on.ca) has immersed himself in objects for more than 10 years and has helped clients with their object immersions as a project mentor and as a course instructor. Wayne (wayne@objectpeople.on.ca) is the coordinator of course construction and a software developer.

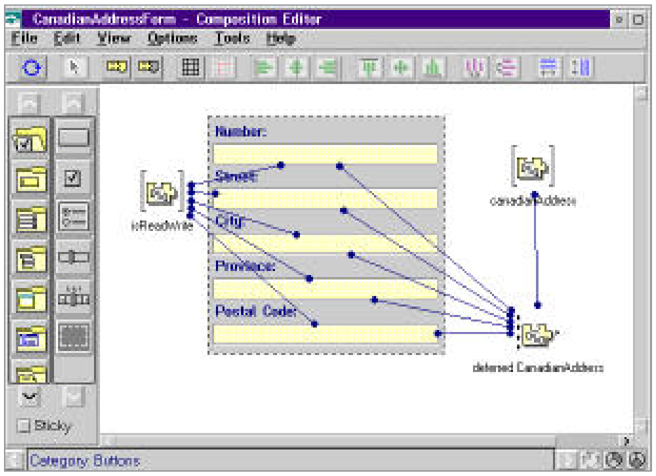

Figure 1. Canadian Address Form.

and modifying an address in the CanadianAddressForm, where it belongs, will also help later to decrease the connections in the Client Profile Editor.

A CanadianAddress is a business object with the five parts: a street number, a street name, a city name, a province name, and a postal code. A CanadianAddressForm, shown in Figure 1, is an editable view for the address. Since a CanadianAddressForm requires an address to edit, one variable part, called 'canadianAddress', is required to reference the address. Also, since the form can be used for both viewing and editing, another variable part, called isReadWrite, is required to store a Boolean, which indicates whether the form is for editing or strictly for viewing. To enable or disable editing in the form's Text parts, the 'self' attribute of the isReadWrite variable part is connected to the 'enabled' attribute of every Text part. To support the undoing of address changes, a Deferred Update part—a VisualAge supplied part— is created from the canadianAddress variable part. A connection between its 'target' attribute and the valueHolder attribute of the variable links the two.

One can view a Deferred Update part as a copy of an original that maintains a stack of changes, which can be applied backwards and forwards to the original part. The Deferred part's interface is similar to the original, having the same attributes, and connects to other parts in an identical manner. In our CanadianAddressForm, we connected the CanadianAddress's attributes to the 'object' attribute on the corresponding Text parts. To permit parts using the CanadianAddressForm to role back or apply the changes from the Deferred Update part to the canadianAddress, its apply and cancel actions are promoted as applyChanges and cancelChanges actions of the CanadianAddressForm.

The isReadWrite and canadianAddress variables are assigned objects by other parts using the CanadianAddressForm. However, to those using the CanadianAddressForm, the isReadWrite and canadianAddress variables appear only as attributes of the form and not as variables, and the connections between these variables and other parts in the form are invisible. To achieve this effect in VisualAge, the variables' self attributes are promoted as isReadWrite and canadianAddress, respectively.

The part is now complete. It has two public attributes: isReadWrite and canadianAddress, and supports two actions: applyChanges and cancelChanges. Anyone using the part must provide a Boolean value for the isReadWrite attribute and a CanadianAddress object for the canadianAddress attribute. To commit the edit changes to the CanadianAddress object one can invoke the applyChanges actions, and to undo any edit changes one can invoke the cancelChanges action.

The exercise is repeated again, but this time for a ClientProfileForm, and for the same reasons: to provide a single format for the display of a ClientProfile, to encapsulate the logic for viewing and modifying it, and to decrease the number of connections in the Client Profile Editor.

A ClientProfile is a business object with three parts: name, age, and address. A ClientProfileForm, shown in Figure 2, is an editable view of the profile. Like the CanadianAddressForm, the ClientProfileForm requires two variable parts: isReadWrite and clientProfile. The purpose of the isReadWrite variable is identical to the one in the CanadianAddressForm, and it has similar connections. The purpose of the clientProfile variable is to provide a reference to the form's business object. To support the undoing of client profile changes, a Deferred Update part is again used, this time created from the clientProfile variable part.

The Client profile form includes a CanadianAddressForm. To use the CanadianAddressForm, the isReadWrite variable part's self attribute is connected to the CanadianAddressForm's isReadWrite attribute, and the clientProfile part's address attribute is connected to its canadianAddress attribute. These connections provide the CanadianAddressForm with the objects it requires to function—in only two connections!

Finally, we want the ClientProfileForm to support the acceptance or cancellation of edit changes. Unlike the CanadianAddressForm, the files have not been touched at all. Where we promoted its deferred part's corresponding actions, this time we have to write two methods: applyChanges and cancelChanges, and add them to the form's public interface. This means that to accept or cancel the changes on this form is to have both its deferred part and the CanadianAddressForm accept or cancel the changes, which can't be done visually. The code for the scripts are as follows:

```
applyChanges
    (self subpartNamed: 'deferred ClientProfile') apply.
    (self subpartNamed: 'Canadian Address Form')
    performActionNamed: #applyChanges.

cancelChanges
    (self subpartNamed: 'deferred ClientProfile') cancel.
    (self subpartNamed: 'Canadian Address Form')
    performActionNamed: #cancelChanges
```

The part is now complete. It has two public attributes:

isReadWrite and clientProfile, and supports two actions: applyChanges and cancelChanges. Anyone using the part must provide a Boolean value for the isReadWrite attribute and a ClientProfile object for the clientProfile attribute. To commit the edit changes to the ClientProfile object, one can invoke the applyChanges actions. To undo any edit changes one can invoke the cancelChanges action.

We now have the parts required to build a Client Profile Editor, shown in Figure 3: a ClientProfile, a ClientProfileForm, and two buttons to invoke the ClientProfileForm's apply and cancel changes actions. In a finished application, these buttons would be replaced with menu items. However, in this article, we wanted to keep it simple and did not get into a discussion on menus. The Editor's connections are as follows:

ClientProfileForm.clientProfile → clientProfile.self
Read/WriteToggleButton.selection → ClientProfileForm.isReadWrite
AcceptButton.clicked → ClientProfileForm.acceptChanges
CancelButton.clicked → ClientprofileForm.cancelChanges

How many connections are required to edit a ClientProfile? There are four: one to instruct the ClientProfileForm which ClientProfile to work with; one to identify whether one is viewing or editing the profile; and two to apply or cancel the end-user's changes to the existing ClientProfile. One could argue that we would have the same number of connections if we implemented all viewing and modification operations in the ClientProfileEditor itself. We would accept that argument.

However, a more important question to ask here is: "Have we gained anything by layering those connections in our ClientProfileForm and a CanadianAddressForm?" The answer is an overwhelming YES! Our two forms and one editor are easy to understand and maintain. We can reuse our forms in any other windows that need to display or modify CanadianAddresses or ClientProfiles. We have a framework for canceling user changes to business objects.

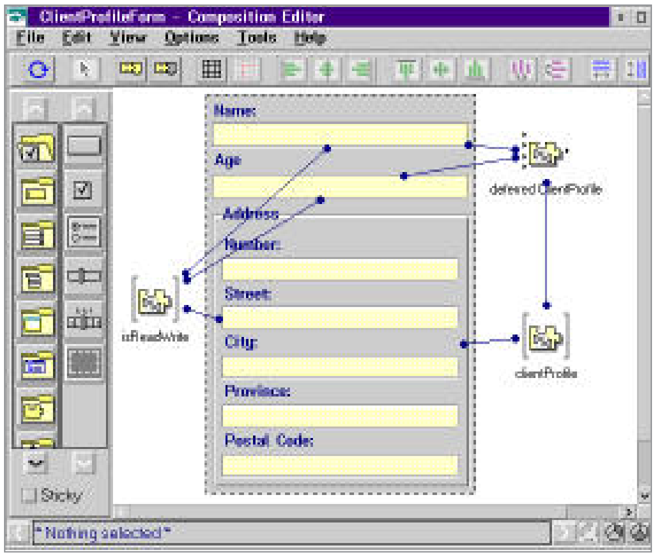

Figure 2. Client Profile Form.

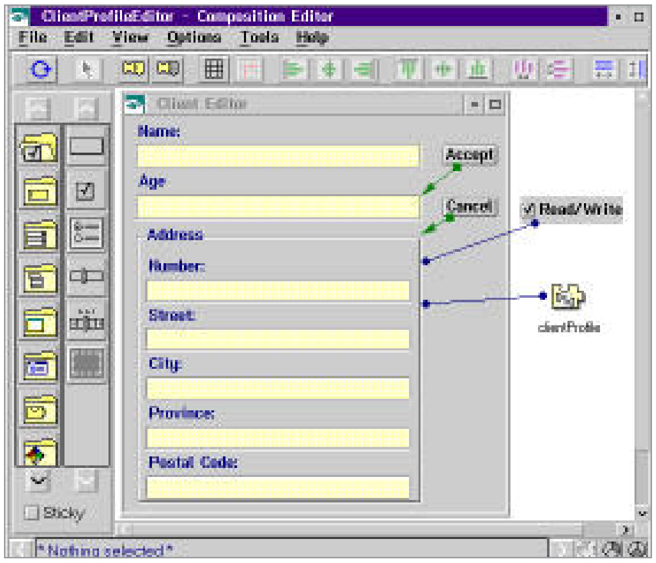

Figure 3. Client Profile Editor.

Finally, we have achieved our original goal of minimizing the connection complexity of our windows.

DO THE RIGHT THING

Minimizing connections and planning for reuse takes some thought. Often, one is required to build a window that manipulates a number of business objects. The temptation is to have the display and modification logic in one window, rather than factor the window into a number of reusable components and use them to construct the window. As seen, factoring your windows into a number of components decreases the number of connections and the complexity of each component, as well as the final window. This makes your components easier to understand and maintain. Even though you may not need the components for any other window yet, you or someone else will! So why not do it the right way to begin with?

Remember, many objects that do little, is better than few objects that do too much. Therefore, a window that is composed of many simple, reusable components, is better than a window that does everything itself. We strongly suggest that every business object have a form built for its display and editing.

Factoring is of course not a new idea. Good GUI developers have been doing it for years with tools that all GUI builders provide. For example, in Visual Works reusable forms are called "subcanvases." In ObjectShare's WindowBuilder they are called "composite panes," and in Digitalk's Parts they are called "nested parts." Whether visually programming or using one of the layout-type GUI builders, building and using reusable forms is not just a good idea, it's great object-oriented programming.

THE CODE

The code presented in this column and in future columns is available on the World Wide Web. Our URL is http://www.objectpeople.on.ca.