SEPARATING APPLICATION CODE FROM TOOLKITS: ELIMINATING THE SPAGHETTI OF CALL-BACKS

Brad A. Myers

School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213

ABSTRACT

Conventional toolkits today require the programmer to attach *call-back* procedures to most buttons, scroll bars, menu items, and other widgets in the interface. These procedures are called by the system when the user operates the widget in order to notify the application of the user's actions. Unfortunately, real interfaces contain hundreds or thousands of widgets, and therefore many call-back procedures, most of which perform trivial tasks, resulting in a maintenance nightmare. This paper describes a system that allows the majority of these procedures to be eliminated. The user interface designer can specify by demonstration many of the desired actions and connections among the widgets, so call-backs are only needed for the most significant application actions. In addition, the call-backs that remain are completely insulated from the widgets, so that the application code is better separated from the user interface.

KEYWORDS: Call-Back Procedures, Dialog Boxes, UIMSs, Interface Builders.

1. Introduction

The Gilt Interface Builder allows dialog boxes and similar user interface windows to be created by selecting widgets from a palette and laying them out using a mouse. More interestingly, Gilt provides a variety of mechanisms to reduce the number of *call-back* procedures that are necessary in graphical interfaces. A ''call-back'' is a procedure defined by the application programmer that is called when a *widget* is operated by the end user. A ''widget'' is an interaction technique such as a menu, button or scroll-bar. A collection of widgets is called a *toolkit*. Examples of toolkits are the Macintosh Toolbox, the Motif and Open-Look toolkits for X windows, and NeXTStep. Most toolkits today require the programmer to specify call-backs for almost every widget in the interface, and some widgets even take more than one call-back. For example, the slider widget in Motif has two call-backs, one for when the indicator is dragged and one for when it is released.

A typical user interface for a moderately complex program will contain hundreds or even thousands of widgets. For example, the VUIT program from DEC uses over 2500 widgets. This means that the programmer must provide many call-back procedures. To add to the complexity, each type of widget may have its own protocol for what parameters are passed to the call-back procedures, and how the procedures access data from the widget.

The use of all of these call-backs means that the user interface code and the application code are not well separated or modularized. In particular:

- The call-backs closely tie the application code to a particular toolkit. Since each toolkit has its own protocol for how the call-backs are called, moving an application from one toolkit to another (e.g., from Motif to Open-Look) can require recoding hundreds of procedures.
- The call-backs make maintaining and changing the user interface very difficult. Changing even a small part of an interface often requires rewriting many procedures. Even if a graphical interface builder is used to change the widgets, the call-backs must be hand-edited afterwards if widgets are added, deleted, or modified.
- The call-backs often are passed the text labels shown to the user, so if the natural language used in the dialog box is changed (e.g., from English to French), the values passed to the call-backs will change, requiring the application code to be edited.

We have observed that many of the call-back procedures are actually used to filter the values from widgets and connect widgets to each other, rather than to perform real application work. By identifying some common tasks that call-backs are used for, and providing other methods for handling the tasks, we have been able to eliminate the need for most call-backs. The tasks can be classified into the following categories:

Preparing the data for applications. Often, call-backs are used to convert the values that the widgets return into a form that the application wants. This may involve converting the type of a value, for example from a string to an enumerated type, or it may involve combining the values from multiple widgets into a single record structure.

Error checking. Before the data is passed to the application, some error checking of it is often needed, along with appropriate messages when there is an error.

Preparing data to be shown to the user. Another set of procedures is usually needed to set the widgets with appropriate default values, which are often dynamically determined by the application. For example, when a color dialog box is displayed, the widgets in it will usually need to be set to the color of the currently selected object. In some cases, it may even be necessary to change the *number* of widgets in the dialog box each time it is displayed, for example, if a button is needed for each application data value.

Internal control. Many call-backs are used to control connections between user interface elements, which require little application intervention. For example, these procedures might cause a widget to be disabled (grey) when a radio button is selected, or cause one dialog box to appear when a button in another is hit.

Gilt provides a standard style of window that allows the filter expressions to be entered. The goal is to minimize the amount of code that needs to be typed to achieve the required transformation. Therefore, much of the filter expression is generated automatically when the designer *demonstrates* the desired behavior. Other parts can be entered by selecting items from menus. As a last resort, the designer can type the required code. If a call to an application function is necessary in a filter expression, Gilt makes sure that the procedure is called with appropriate high-level parameters, rather than such things as a widget pointer or the string labels. Thus, the call-backs that remain are completely insulated from the user interface.

Gilt is a part of the Garnet system [8]. Garnet is a comprehensive user interface development environment containing many high-level tools, including Gilt, the Lapidary interactive design tool [7], the C32 spreadsheet system [9], etc. Garnet also contains a complete toolkit, which uses constraints [15] and a prototype-instance object model. Gilt stands for the Garnet Interface Layout Tool, and it supports interfaces built using either the Garnet look-and-feel widget set or the Motif look-and-feel widget set.[1] Gilt uses CommonLisp, but the ideas presented here are applicable to interface builder tools using conventional compiled languages.

[1]The Motif-style widgets in Garnet are implemented on top of the Garnet Toolkit intrinsics and do not use any of the Xtk code in C. Although they look and behave like the standard Motif widgets, they have the same procedural interface as the Garnet widget set.

2. Related Work

Of course, there are a large number of commercial and research interface builders that lay out widgets, including DialogEditor [3], the Prototyper for the Macintosh [13], the NeXT Interface Builder, UIMX for Motif, and Druid [12]. However, these only have limited mechanisms for reducing call-backs. Many of them support transitions from one dialog box to another, and NeXT allows the output value of one widget to be connected to the input of another, if no filtering is needed. Druid adds the ability to set the initial values for widgets (but only statically, not application data dependent), and to collect values of widgets for use as the parameter to a procedure. It allows the designer to specify some of these by demonstration. However, in Gilt, significantly more of the user interface can be specified without requiring call-backs, the call-backs are more independent of the widgets, and a uniform framework is used for all filtering.

A primary influence on Gilt is the Peridot UIMS [6]. Peridot was the first system to allow the designer to specify the behavior of the interface by demonstration. Gilt uses some of the techniques in Peridot to guess the appropriate transformations based on the example values.

The filter expressions that the designer specifies in Gilt are implemented using *constraints*. A constraint is a relationship that is declared once and then maintained by the system. Constraints have been used by many systems, starting with Sketchpad [14] and Thinglab [ThinglabToplas]. Uses of constraints within user interface toolkits include GROW [1], Peridot [6], and Apogee [5].

Other systems have allowed the designer to specify the connections between the user interface and the application procedures at a high level. The Mickey system [11] uses special comments in the procedure definition to describe the connection to the user interface. The UIDE system [4] allows the application procedures to be defined in advance, and generates the interface partially from these. Unlike these systems, Gilt requires the designer to specify the graphics, and then explicitly attach the graphics to the procedures, but it infers the mapping between the values returned by the widgets and the values desired by the procedures.

3. Example

To show how easy it is to define dependencies without writing call-backs, we will first present an example of creating the dialog box of Figure 1. There are a number of dependencies in this relatively simple interface. The return value of the dialog box is a font object. If one of the standard fonts is selected, then the corresponding built-in font object should be returned. Otherwise, the return value will be a font specified by name, so the specified file should be opened, and a new font object created for that file.

First, the user would create the graphics for the dialog box by selecting the widgets from a palette and typing in the correct labels, in the conventional direct manipulation man-

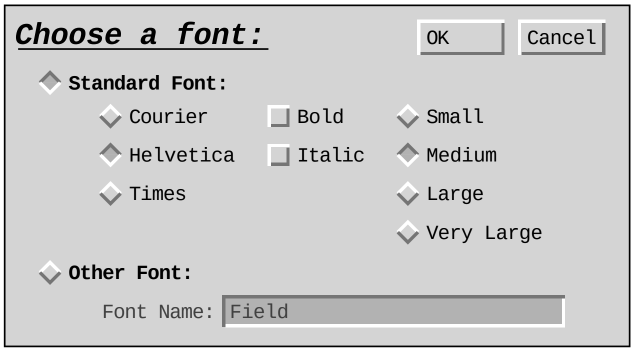
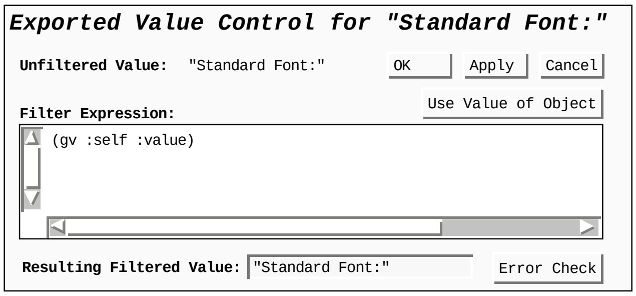


Figure 1:
A font selection dialog box being created in Gilt. When the `Standard Font` radio button is pressed, the `Font Name` field is disabled (grey), and when the `Other Font` radio button is selected, the three sets of buttons under `Standard Font` (for the family, face and size of the font) become disabled.

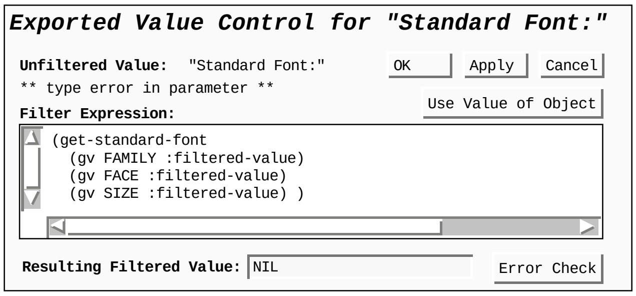
ner as with other interface builders. Next, the designer gives meaningful names to the widgets (e.g., the `Bold/Italic` radio buttons are called "face").

The default value of a set of radio buttons is the string label of the selected button. We will now override this and make the value of the `Standard Font` branch instead be the appropriate font object. To do this, we bring up the Gilt window in Figure 2-a. When it is brought up, it initially shows that the resulting exported value from the widget is the same as the widget's value. To get the appropriate font object, we need to call the Gilt function `get-standard-font`, so we choose this from a menu. This inserts the function call into the filter expression. The procedure should be passed the values of the three sets of widgets under `Standard Font`. Therefore, we select the three widget sets and hit the `Use-Value-of-Object` button in the window. This inserts references to all the selected objects into the filter expression, resulting in Figure 2-b. The references are inserted in the order the objects were selected. These references will be to the *filtered* values of the widgets, which so far are the same as the default values: the string names of the labels. However, Gilt knows that `get-standard-font` expects Lisp keywords as arguments rather than strings (a "keyword" is an atom prefixed by a colon, such as `:bold`). Therefore, Gilt can tell that there is a mismatch, so it tries to determine a possible transformation. Another `Exported Value Control` window pops up for each of the selected widgets, and the designer can check that the inferred transformations are correct (Figure 2-c). If not, the designer can give additional examples or explicitly edit the generated code. In this example, however, the system guesses all cases correctly, so the designer simply hits "OK" on all of the windows. This will assert constraints so that the filtered values of the widgets will be keywords, as required.

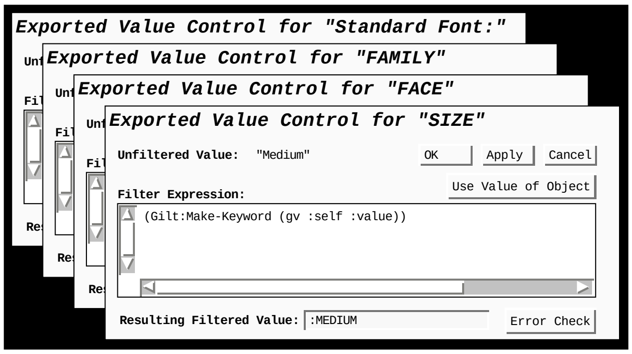
Now, the value for the other branch must be set. The designer selects the `Other Font` radio button and brings up the `Exported Value Control` window for it. By selecting the `get-font-from-file` function from a



(a)



(b)



(c)

Figure 2:
(a) The Gilt window that allows the designer to control how values for widgets are filtered. Many of the fields are filled in by Gilt as the designer demonstrates the desired behavior. The `Unfiltered Value` shows the value as currently provided by the widget before any filtering. The `Filter Expression` is the Lisp expression to filter the value. The designer can hit the `Use Value of Object` button to insert a reference to the value of a selected object. The default filter simply copies the original value. The `Resulting Filtered Value` field shows the final value after the filtering. This field can be edited to show the transformation for the current widget by example. (b) shows the filter expression after a function has been selected from a menu and the widget references have been filled in. (c) shows the additional windows that appear to confirm the transformations that are inferred for the widgets that are referenced in (b).

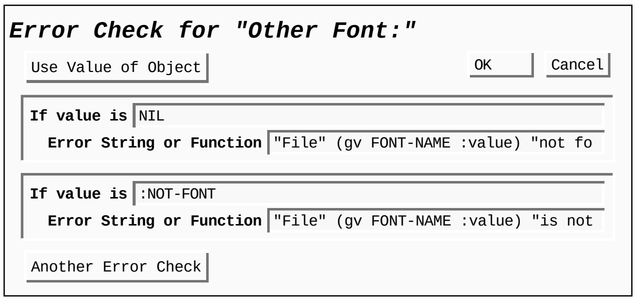


Figure 3:
This window allows the designer to specify the handling of error values. When `Other Font`'s filtered value is NIL, the first error string is printed, and when `Other Font` is the special value `:NOT-FONT`, the second string is printed. The `Use Value of Object` button is used to insert a reference to a selected object, here, the value of the `Font Name` widget, which contains the current file name. The `Another Error Check` button causes another `If value is` and `Error String` pair to appear.

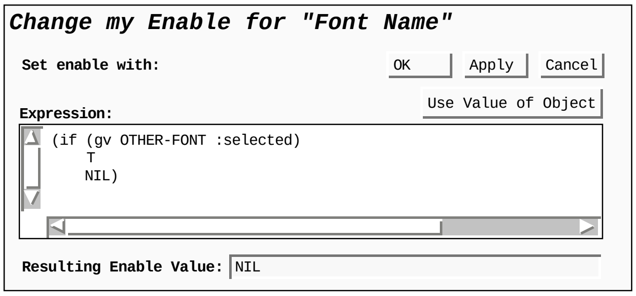


Figure 4:
The Gilt window that allows the designer to specify that the enable property of a widget depends on other widgets. When the `Expression` returns NIL, the widget is shown ''greyed-out.''

menu, then selecting the `Font Name` widget, and finally hitting the `Use Value of Object` button, the designer can specify the appropriate dependencies. Since `get-font-from-file` expects a string, no further transformations are needed. If the font is not found, the `get-font-from-file` function returns error values, so the `Error Check` window is used to specify the handling of this (Figure 3). The designer types the appropriate error return values and response strings into this window.

Next, the value of the entire dialog box is specified as the value of the pair of radio-buttons `Standard Font` and `Other Font`, and now the dialog box will return a single value, computed based on the settings of the widgets.

Finally, the designer needs to specify when the various widgets should be disabled (greyed out). First, the designer selects the `Font Name` text field, and then brings up the `Change my Enable` window (see Figure 4). Note that this window has the same general form as the Value Control window, but simply controls a different property of the widgets (the enable flag rather than the value). Next, the designer selects the `Other Font` radio button and hits the `Use Value Of Object` button. This makes the `Font Name` enabled (not grey) when `Other Font` is chosen. Similarly, the family, face and size buttons under `Standard Font` are enabled when `Standard Font` is selected.

4. Filtering

Each widget in Garnet will always first compute its default value, which is then assigned to the widget's slot (instance variable) called `:VALUE`.[2] This value can then be filtered to derive the value seen by application programs, which is set into the slot called `:FILTERED-VALUE`. This is implemented as a constraint that sets the value of the `:FILTERED-VALUE` slot whenever the value of the `:VALUE` slot changes. The default constraint simply copies the value. Experience has shown that most filter expressions are rather short, often only one or two lines. Sometimes, it will be necessary to have longer, complex transformations or access to application-specific functionality and data. Here, a conventional text editor would be used to create a function which will then be called by the filter expression entered with Gilt. However, the function will be independent of the particular widgets used since Gilt provides transformations of the arguments and return values from the function.

As was shown in the example, Gilt provides a number of ways to specify the appropriate filtering of data and control in the user interface, so the application code is independent of the particular widgets used and the label strings shown to the user. All of these transformations use the same, standard `Control` windows shown in the previous examples. The following sections show how the various tasks that require call-backs in other toolkits are performed in Gilt.

4.1 Preparing Data for Applications

Many call-backs in widgets simply filter the output value to convert it to a form needed by the application program. For example, for Figure 1, you might need as many as 13 different call-backs in other toolkits to generate the single font value to be returned. In Gilt, the value of the dialog box is available in a variable, without requiring a call-back.

Unlike most toolkits, Garnet provides values for *groups* of widgets. For example, the default value of a radio button set is the name of the radio button that is selected, or NIL if none are. For a set of check boxes (that allows multiple selections), the value is a list of the selected buttons. The innovation in Gilt is that the designer can specify alternative values for widgets. In the example, the value of the pair of radio buttons `Standard Font/Other Font` will be a font object.

Many of the desired transformations of the values can be achieved by simple type conversions: from strings to keywords, atoms, numbers, etc. Therefore, Gilt provides a

[2] All slot names in Garnet start with a colon.

number of built-in data transformations:

- String to Lisp atom (e.g. `"Bold"` to `'BOLD`).
- String to Lisp keyword (e.g. `"Bold"` to `:BOLD`).
- String to index of item in the set of buttons (e.g. `"Bold"` to 0).
- String to number (e.g. `"10"` to `10`).
- Integer range to a different integer or float range.

Similar transformations would be appropriate for a builder generating other computer languages, like C or Pascal, which might automatically create enumerated types, sets, bit vectors, or named constants.

Gilt tries to automatically pick the appropriate transformation. There are two techniques used to guess what is appropriate.

First, the designer can type an example value into the `Resulting Filtered Value` field at the bottom of the `Exported Value Control` window (Figure 2-a). In this case, Gilt will try to guess a transformation that will convert the current unfiltered value into the specified value, using the above rules. If none of the built-in transformations is appropriate, then Gilt creates a `case` statement. The designer can then operate the widget to put it into different states (and therefore to change the unfiltered value), and type the desired filtered value for each case. This allows arbitrary transformations (e.g., converting the German `"Fettdruck"` or the French `"Gras"` to `:BOLD`). The resulting code for the filter is shown in the `Filter Expression` window.

The second option is used when the designer enters a procedure into the filter expression, and then selects a widget to supply the value to a parameter of the procedure. Here, Gilt tries to find an appropriate transformation so that the widget value will be filtered into the required type of the parameter. This is the technique used in the example. A `Value Control` window will pop up to confirm each transformation, and also to request the designer to specify the transformation if Gilt cannot infer it.

A number of standard procedures are provided in a pop-up menu, so the designer can often select a procedure for the filter expression rather than typing it. The provided routines will transform a string into a file pointer, a string into a font pointer, numbers or a string into a color, keywords into a font, etc. If one of these is selected from the menu, the appropriate code is entered into the `Filter Expression` field. Because these routines take abstract values as parameters, and return a value of the appropriate type (such as a font object), the implementation of the routines is entirely independent of the widgets. In fact, standard, built-in routines, such as the Lisp function `probe-file`, can be used in many cases.

Gilt can execute the filter expressions, including any procedures entered by the designer, by using the Lisp interpreter. Therefore, when Gilt is put in ‘‘run-mode’’ the actions will happen just as they will for the end user. Gilt first checks to make sure that all procedures are defined, in case the designer has entered an application-specific procedure that is not implemented yet. In this situation, Gilt

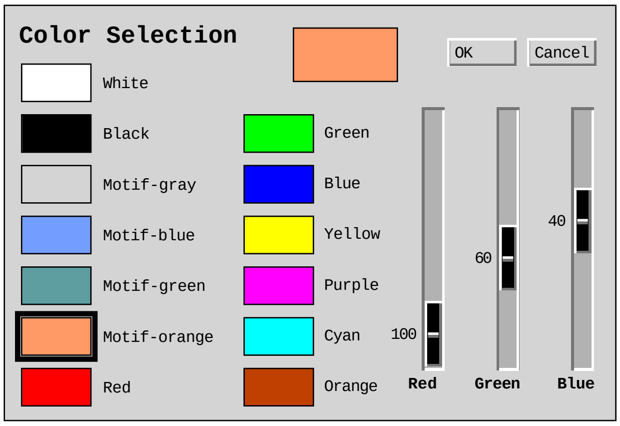


Figure 5:
The color selection dialog box created using Gilt (naturally, this is in color on the screen). There are a number of dependencies among the widgets that were defined by demonstration. If one of the color buttons on the left is selected, the sliders adjust to the appropriate position for that color. If the sliders are moved, the highlight in the color buttons (here shown around `Motif-orange`), goes to the appropriate color or goes off. The rectangle in the upper center always shows the current color. The filtered value of the rectangle is its color, and the value of the dialog box is defined as the filtered value of the rectangle.

requests the designer to give an example of the value the function would return.

Sometimes, the value of a widget might be computed based on the values of *multiple* other widgets. In the example of section 3, the value of the `Standard Font` radio button is computed based on the values of three sets of buttons. The default expression creates a list out of the values, but by editing the filter expression, it is easy to create a record or structure instead of a list, or to process the values in various ways. In Figure 2-b, the `get-standard-font` routine is called on the values of three widgets to return a single font object.

Gilt allows decorations to be added to the dialog boxes, such as rectangles, lines and labels. These normally do not have a value, but they can be given one using a `Value Control` window. For example, the rectangle at the upper center of Figure 5 shows the current selected color. The value of this rectangle should be its color. To achieve this, the designer can type `(gv :self :COLOR)` into the `Filter Expression` field.[3] To make this a little easier, the designer can choose the desired field of the selected object from a pop-up menu.

The user can check that the filter expression is achieving the desired result in two ways. First, the interface can be exercised to test the code. Second, the `Filter Expression` field shows the Lisp code that is being used.

[3] `gv` stands for ‘‘get value’’ and it looks in the specified object for the specified slot.

In the future, we will be investigating other techniques for showing the transformations that will be usable by non-programmers. For example, the filter expressions might use normal arithmetic expressions, or we might create a special graphical programming language.

4.2 Error Handling

Call-back procedures in other toolkits are often used to check for error values, especially in text input fields. Gilt provides a standard error-filtering mechanism that minimizes the connections between the error checking code and the widgets. The designer can bring up the `Error Check` window (Figure 3), and type a value into the `if-value-is` field. If the filtered value for the widget is ever equal to the `if-value-is` value, then an error has occurred. If the `Error String` field contains a string, then a error dialog box is popped-up showing that string. The string can embed references to other widgets using the `Use-Value-of-Object` button, for example, to show the incorrect value. Alternatively, if the `Error String` field contains an expression or function call, then it is executed.

Alternatively, an expression using the value of the widget can be entered into the `if-value-is` field, which should return T if an error should be reported. For example, to report an error if an input number is odd, the designer could simply enter `(oddp (gv :self :FILTERED-VALUE))`. If the filter expression itself returns an error message string, then the `if-value-is` might just test if the filtered value is a string, and the `Error String` would just be `(gv :self :FILTERED-VALUE)`.

There can be multiple `if-value-is` and `Error String` pairs, which would be useful, for example, for a font finding routine that returned different values to tell if the file was not found, or if the file was not a valid font, as in Figure 3. The `get-font-from-file` filter will return a font, or NIL if the file is not found, or `:NOT-FONT` if the file is found, but it is not a font.

4.3 Preparing Data to be Shown to the User

Most toolkits require that the designer create additional procedures to *set* the widgets based on application-specific data. For example, when many dialog boxes are made visible, the values of some widgets should be set to a particular value. If a widget should *always* have the same value when the dialog box appears, then the designer can simply supply this value by example, as in other interface builders like Druid [12]. However, it is very common for the initial values for widgets to depend on application-supplied data. For example, when the font dialog box is made visible by an application, it should reflect the font of the selected object, or if there is no object selected, then the current global default. The next sections discuss how Gilt allows this to be specified easily.

4.3.1 Defining Parameters to the Dialog Box

When a window is designed in Gilt, parameters to the window can be specified, along with an example current value

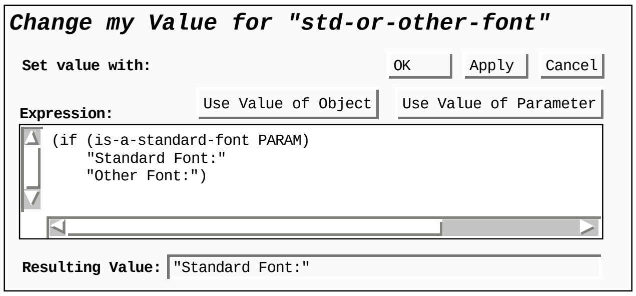


Figure 6:
The Gilt window to cause the displayed value of a widget to change based on other widgets. Here, the `Standard Font/Other Font` radio buttons of Figure 1 are set based on the value of the parameter. The designer only had to select the `is-a-standard-font` procedure from a menu, the rest of the expression was entered by Gilt as the widgets in Figure 1 were operated.

for the parameter. If an application wants to display a window designed in Gilt, it can simply call[4]

`(Show-Dialog` *dialog-name param1 param2* `...)`

For example, the font dialog box of Figure 1 would take a single font object as a parameter. Thus, the application causes the dialog box to appear while still being independent of how the parameters are used to set the widgets.

For ‘‘modal’’ dialog boxes (that require the user to say OK or CANCEL before doing other operations), the `Show-Dialog` routine will return the value of the dialog box. The designer can specify the value of the dialog box using a `Value Control` window, as was shown in the example. For non-modal dialog boxes, `Show-Dialog` will return immediately, and the designer can attach a call-back procedure to the OK button. Of course, this call-back will be passed the filtered value of the window, so it will be independent of the widgets that are used in the window to enter the value.

4.3.2 Using the Parameters

To set the value of a widget based on the parameters, the designer uses the `Change my Value` window (see Figure 6). The primary difference from the `Value Control` window shown earlier is that here we are changing the value shown to the user, rather than simply filtering the value returned by the widget. However, this window is very similar to the `Value Control` window, and the interface to the designer is essentially the same.

The result of the expression should be an appropriate value for the widget. For example, Figure 6 calculates the string

[4] In a language that does not support functions with a variable number of arguments, a Gilt-like builder could create a different `show-<dialog-name>` routine for each window designed.

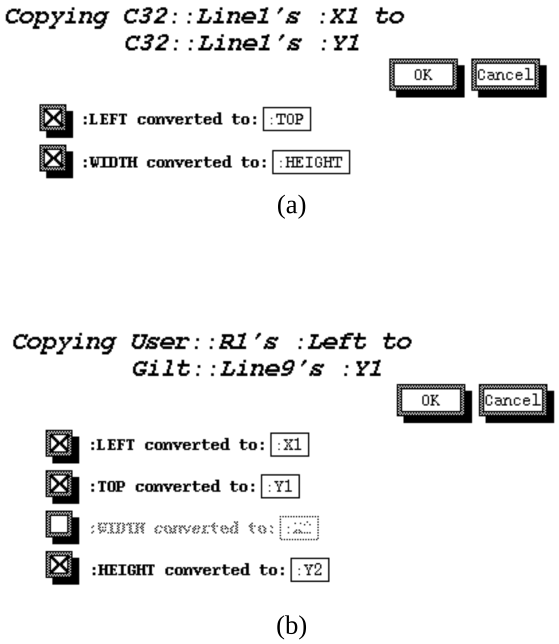


Figure 7:
This dialog box (which uses the Garnet widget set instead of the Motif widget set used by the other figures in this paper), repeats the check box, the label and the text type-in field. The controlling expression for (a) might be `((T :LEFT :TOP) (T :WIDTH :HEIGHT))`, where the T controls the radio button, the second element is used in the label, and the third is used as the default for the text input field. The user can then turn on and off the desired slots using the check boxes, or type a new name.

name of the branch of the radio button to be selected. Of course, designers can simply type in the appropriate code, but Gilt provides demonstrational techniques to make this easier. The designer can operate the widgets to put them into the appropriate state, and then give the expression that will determine when that state is to be used. For example, for the font dialog box, the designer could select the `Standard Font/Other Font` widget, and bring up a `Change my Value` window (Figure 6). Then, the `Standard Font` radio button would be pressed, and the designer could hit the `Use Value of Parameter` button. Then, the designer would have to edit the expression to return T when the font was a standard font using the `is-a-standard-font` procedure. By default, the other value of the radio buttons will be used otherwise, so nothing is needed for that case. Next, the designer would bring up `Change my Value` windows for the other widgets, such as `Font Name`, and write expressions to extract the appropriate information from the font object parameter.

4.3.3 Dynamic Creation of Widgets

Sometimes, a parameter might specify the *number* of widgets that need to be created. In this case, the designer can show by example the set of widgets to be replicated, select them, and bring up a `Replicate Control` window, which is similar to the `Change my Value` window. The expression in this window is expected to return an integer to tell how many copies of the widgets are desired. Alternatively, the expression can return a list of values, in which case, the number of copies depends on the length of the list. Here, each copy is assigned the appropriate element from the list. For example, in Figure 7, the application might supply as a parameter to the dialog box a list of slot names to control how many times the check box, the label and the text input field are repeated.

4.4 Internal Control

In other toolkits, another set of call-back procedures are often needed to control the setting of the value or other property of one widget based on the value of another, or to bring up a new dialog box when a button is pressed. The next sections discuss how Gilt allows these to be specified using filter expressions.

4.4.1 Value Dependencies

Sometimes, when a widget is operated, this should cause a different widget to change its value. For example, when the user hits on a color button in Figure 5, the sliders should move to show the appropriate values for that color. Gilt provides a convenient mechanism for specifying this using the same `Change my Value` window as for having a widget's value depend on a parameter (Figure 6).

The designer selects the widget that should change (for example, the red slider of Figure 5), and brings up a `Change my Value` window. Next, the widget that it should depend on (here, the color button set) is selected, and the `Use-Value-of-Object` button is hit. This will generate the expression

```
(gv Color-buttons :FILTERED-VALUE)
```

but for the red slider, only the red component of the color should be used, so the designer would edit the expression to be

```
(gv Color-buttons :FILTERED-VALUE :RED)
```

Now, whenever the color buttons are operated, the red slider will be set correctly. The other two sliders would be fixed similarly.

Sometimes, widgets may need to be replicated based on the value of another widget. In the Xerox Star and Viewpoint, menus only show legal values, rather than greying out illegal values. For example, in a font-choice dialog box, if different fonts have different sizes available, the components in the menu of sizes must be dynamically changed. The `Replicate Control` window discussed in section 4.3.3 is used to control this.

4.4.2 Specifying Other Dependencies

The previous sections discussed how the *value* of a widget can be controlled. In many cases, however, other properties of widgets may need to be set, such as whether it is enabled or not (greyed-out). This is handled in a uniform way, using a window similar to the `Change my Value` window. The designer selects the widget to controlled, specifies the desired property from a menu, and the appropriate window is brought up.

4.4.2.1 Enabling

One of the most common dependencies is to enable widgets based on other widgets. As shown in the example of section 3, the designer can operate a widget to have the appropriate value, then enable or disable the dependent widget, and Gilt will fill in the values for the `Change my Enable` (Figure 4). In trying to guess appropriate control expressions for dependent slots, Gilt knows about check boxes and radio buttons being on or off, text fields being empty or having a value, and numbers being zero or non-zero. In addition, if the `Change my Enable` window is for a *set* of selectable items (such as a menu or a panel of buttons), the controlling widget can return a list of values, each element of which controls an item. For example, in Figure 8, the menu of font sizes will have a `Change My Enable` expression that computes the list of valid font sizes based on the selected font in the left menu. Although an application function is needed, the function will be independent of the particular widgets used, since it will take a font object and return a list of valid sizes. Gilt will automatically create an expression to enable the items that correspond to the values in the list and disable the others.

4.4.2.2 Other Properties

All the other properties of widgets can be controlled in the same way as enabling. Widgets can be made to be visible and invisible by bringing up a `Change my Visible` window. Most widgets also have additional properties which can be set, such as their color or font. To change the color of an object, the `Change my Color` window is used. For example, to change the color of the red slider based on the value it returns, the designer could simply select the red slider, bring up the `Change my Color` window, select the slider again, hit the `Use-Value-of-Object` button, and then edit the expression to be[5]

```
(Make-Color (gv :self :FILTERED-VALUE)
            0 0)
```

Using the dependency control on various properties is also useful for decorations such as rectangles and labels. For example, the color of the rectangle in the center of Figure 5 can be made to depend on the three sliders in this way.

4.4.3 Sequencing of Dialogs

Another common internal control action that sometimes requires call-backs is for a button to cause another dialog box to appear. Gilt, like other interface builders, allows this to be demonstrated, by simply operating the button, and showing which dialog box should appear. However, unlike other systems, Gilt also allows the initial values of widgets in the sub-dialog to be set. Windows similar to the `Change my ...` windows appear that allow the values of the parameters to the sub-dialog to be specified based on the values of the parent dialog box. Gilt will automatically create the code to call `Show-Dialog` in the appropriate way. If the sub-dialog is modal (which is the usual case), then the value of the sub-dialog is assigned by default as

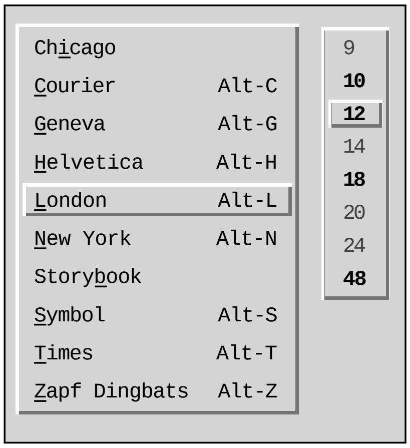


Figure 8:
In these Motif-style menus, the various font sizes in the menu on the right become enabled or disabled depending on the sizes available for the font that is selected in the menu on the left.

the value of the button that caused the sub-dialog to be displayed. Of course, the designer can control this using the `Value Control` for the button.

If the sub-dialog is not modal, then the end user will be allowed to operate widgets in both windows. Gilt supports cross-window dependencies, so that a value in one dialog box can depend on a value in another dialog box.

5. Editing and Saving

To edit the value of any of the filter expressions for a widget, the designer can simply select the widget and bring up the appropriate `Control...` or `Change my...` window. The designer can then edit the text of the expression. Alternatively, if the user demonstrates new transformations, these will replace the existing ones as appropriate.

Gilt provides a special feature to make it easier to convert an interface to a different natural language. After a value transformation has been specified, the next time the designer edits the displayed label names, Gilt will pop-up a question to ask if the corresponding exported values should change also. If the designer says "no", then the value filter function is automatically changed so that all the new label strings will still produce the same old values, so any code that uses the values will not need to be changed.

Other special features make editing the widgets easier. Gilt provides a ‘‘Replace widget’’ command, which allows, for example, a set of buttons to be replaced by a menu. As many of the properties as possible are retained, including the label names and the filter expressions. In addition, the filter expressions can be copied from one widget to another. Finally, because the more complex filter procedures and application-specific call-backs are called with abstract parameters (such as keywords), they usually will not need to be changed when the widgets are edited. We will be investigating other techniques for editing in the future.

---

[5]The slider's value is a number, but we need a color object for the color property. `Make-Color` is a standard routine that takes numbers representing the red, green and blue values and returns a color object.

As was mentioned previously, the expressions are implemented as constraints attached to the appropriate properties of objects. Garnet has a built-in mechanism for saving any object as a Lisp code file, including all of its constraints [10], and this is used by Gilt. Therefore, all the filter expressions are output automatically along with the user interface definition. Since the output is textual Lisp code, it is possible for programmers to edit the file directly, but we expect this to not be necessary.

6. New Kinds of Widgets

The techniques that have been described are not limited to only the built-in widgets in the Garnet toolkits. If the user wants a new kind of widget, then it can be created either by coding it by hand or using the Lapidary design tool [7]. The new widget can then be dynamically loaded into the Gilt palette, and used like any built-in widget.

All of the widgets in the Garnet toolkit are controlled through the same protocol, which includes a specification of what the properties of the widget are and the types of the properties (string, boolean, integer, list, etc.). This allows the appropriate `Control` windows to be created. For custom widgets, the designer will need to conform to the standard protocol. Lapidary has built-in mechanisms to help with this for widgets created using it. The inferencing of the filter expressions is based on the type of the properties, so the demonstrational techniques described in this paper can be used for designer-created widgets as well. As an example, the color selection buttons on the left of Figure 5 are not a standard widget, but were partially coded by hand and then read into Gilt for the dependencies to be specified.

Another interesting feature is that a set of widgets can be saved, along with their interdependencies defined in Gilt, and used as a prototype in other interfaces. For example, the `Standard Font` group from Figure 1 could be read into the Gilt palette, and then placed in other dialog boxes. Due to the prototype-instance object model in Garnet, no extra mechanisms are needed in Gilt to support this.

7. Status and Future Work

An earlier version of Gilt has been released to all Garnet users.[6] The version described here has been mostly implemented, and is expected to be finished and released in the next few months.

In the future, in addition to releasing this version of Gilt for general use, we would like to investigate combining some of the features of Lapidary with Gilt, so that the designer can specify constraints on the widgets, for example to make decorations or the entire window grow if a widget gets bigger. It has been suggested that a wiring diagram approach to specifying the interdependencies among widgets might be easier to use. We will investigate allowing the designer to draw wires among the widgets to show the flow of values and enabling. This might also be helpful as a debugging tool to show where the dependencies are. Other debugging and maintenance aids will also be added, such as browsers to show all the filter expressions, and the procedures and global variables used in them. Finally, we will add some of the demonstrational techniques from Peridot and Druid that neaten the display as widgets are drawn.

8. Conclusion

The Gilt interface builder contains an number of innovations that significantly improve the separation of application code from toolkits. By identifying the most common tasks that call-backs are used for, Gilt is able to supply built-in mechanisms to handle them. Using a standard style of window, the designer can enter short filter expressions. Because many of the tasks involve straightforward filtering, Gilt can often infer appropriate transformations from examples of the desired output or actions. Even when more complex transformations are required, and which use application-specific procedures, the application code is completely independent of the actual widgets and the names used in the user interface. Although Gilt is implemented in Lisp, which makes the dynamic execution of the entered code much easier, the general techniques are appropriate for conventional compiled languages and for interface builders for conventional toolkits. Therefore, the techniques could be readily applied to today's user interface tools.

The mechanisms that are described here make it much faster to build dialog boxes with interdependencies among the widgets. However, we expect their main advantage to be the improved maintainability of the resulting code. For example, it should be much easier with Gilt than most other interface builders to convert a user interface to a different natural language or switch between different forms of widgets (e.g., from menus to buttons), or even different widget sets (e.g., from Motif to OpenLook). We will be exploring the effects of these features as Gilt becomes widely used by the Garnet community.

Acknowledgements

Andrew Mickish implemented the features described in this article. Osamu Hashimoto also contributed to the design and implementation of Gilt. Brad Vander Zanden, David Kosbie, Andrew Mickish, Osamu Hashimoto, Bernita Myers, and the referees provided useful comments on this paper.

This research was sponsored by the Avionics Lab, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, Arpa Order No. 7597.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

---

[6] The Garnet system is available for free from CMU, but you need to have a license. If you are interested in using Gilt and Garnet, please contact the author or send electronic mail to `garnet@cs.cmu.edu`.

References

1. Paul Barth. "An Object-Oriented Approach to Graphical Interfaces". *ACM Transactions on Graphics 5*, 2 (April 1986), 142-172.

2. Alan Borning. "The Programming Language Aspects of Thinglab; a Constraint-Oriented Simulation Laboratory". *ACM Transactions on Programming Languages and Systems 3*, 4 (Oct. 1981), 353-387.

3. Luca Cardelli. Building User Interfaces by Direct Manipulation. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'88, Banff, Alberta, Canada, Oct., 1988, pp. 152-166.

4. James D. Foley, Christina Gibbs, Won Chul Kim, and Srdjan Kovacevic. A Knowledge-Based User Interface Management System. Human Factors in Computing Systems, Proceedings SIGCHI'88, Washington, D.C., May, 1988, pp. 67-72.

5. Tyson R. Henry and Scott E. Hudson. Using Active Data in a UIMS. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'88, Banff, Alberta, Canada, Oct., 1988, pp. 167-178.

6. Brad A. Myers. *Creating User Interfaces by Demonstration.* Academic Press, Boston, 1988.

7. Brad A. Myers, Brad Vander Zanden, and Roger B. Dannenberg. Creating Graphical Interactive Application Objects by Demonstration. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'89, Williamsburg, VA, Nov., 1989, pp. 95-104.

8. Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Ed Pervin, Andrew Mickish, and Philippe Marchal. "Comprehensive Support for Graphical, Highly-Interactive User Interfaces: The Garnet User Interface Development Environment". *IEEE Computer 23,* 11 (Nov. 1990), 71-85.

9. Brad A. Myers. Graphical Techniques in a Spreadsheet for Specifying User Interfaces. Human Factors in Computing Systems, Proceedings SIGCHI'91, New Orleans, LA, April, 1991, pp. 243-249.

10. Brad A. Myers and Brad Vander Zanden. "An Environment for Rapid Creation of Interactive Design Tools". *The Visual Computer; International Journal of Computer Graphics* (1991), to appear.

11. Dan R. Olsen, Jr. A Programming Language Basis for User Interface Management. Human Factors in Computing Systems, Proceedings SIGCHI'89, Austin, TX, April, 1989, pp. 171-176.

12. Gurminder Singh, Chun Hong Kok, and Teng Ye Ngan. Druid: A System for Demonstrational Rapid User Interface Development. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'90, Snowbird, Utah, Oct., 1990, pp. 167-177.

13. SmethersBarnes, P.O. Box 639, Portland, Oregon 97207, Phone (503) 274-7179. Prototyper 3.0.

14. Ivan E. Sutherland. SketchPad: A Man-Machine Graphical Communication System. AFIPS Spring Joint Computer Conference, 1963, pp. 329-346.

15. Brad Vander Zanden, Brad A. Myers, Dario Giuse and Pedro Szekely. The Importance of Indirect References in Constraint Models. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'91, Hilton Head, SC, Nov., 1991.

Table of Contents

1. Introduction 0
2. Related Work 1
3. Example 1
4. Filtering 3
    4.1 Preparing Data for Applications 3
    4.2 Error Handling 5
    4.3 Preparing Data to be Shown to the User 5
        4.3.1 Defining Parameters to the Dialog Box 5
        4.3.2 Using the Parameters 5
        4.3.3 Dynamic Creation of Widgets 6
    4.4 Internal Control 6
        4.4.1 Value Dependencies 6
        4.4.2 Specifying Other Dependencies 6
            4.4.2.1 Enabling 7
            4.4.2.2 Other Properties 7
        4.4.3 Sequencing of Dialogs 7
5. Editing and Saving 7
6. New Kinds of Widgets 8
7. Status and Future Work 8
8. Conclusion 8
Acknowledgements 8
References 9

List of Figures

Figure 1:	**2**
Figure 2:	**2**
Figure 3:	**3**
Figure 4:	**3**
Figure 5:	**4**
Figure 6:	**5**
Figure 7:	**6**
Figure 8:	**7**