Graphical Styles for Building User Interfaces by Demonstration

Osamu Hashimoto
C&C Systems Research Labs.
NEC Corporation
4-1-1 Miyazaki, Miyamae-Ku,
Kawasaki, Kanagawa 216, Japan
osamu@tsl.cl.nec.co.jp

Brad A.Myers
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue,
Pittsburgh,PA 15213
brad.myers@cs.cmu.edu

ABSTRACT
Conventional interface builders allow the user interface designer to select widgets such as menus, buttons and scroll bars, and lay them out using a mouse. Although these are conceptually simple to use, in practice there are a number of problems. First, a typical widget will have dozens of properties which the designer might change. Insuring that these properties are consistent across multiple widgets in a dialog box and multiple dialog boxes in an application can be very difficult. Second, if the designer wants to change the properties, each widget must be edited individually. Third, getting the widgets laid out appropriately in a dialog box can be tedious. Grids and alignment commands are not sufficient. This paper describes *Graphical Tabs* and *Graphical Styles* in the Gilt interface builder which solve all of these problems. A "graphical tab" is an absolute position in a window. A "graphical style" incorporates both property and layout information, and can be defined by example, named, applied to other widgets, edited, saved to a file, and read from a file. If a graphical style is edited, then all widgets defined using that style are modified. In addition, because appropriate styles are inferred, they do not have to be explicitly applied.

KEYWORDS: User Interface Builder, User Interface Management System, Demonstrational Interfaces, Styles, Tabs, Garnet, Direct Manipulation, Inferencing

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 by direct manipulation (see Figure 1). Two sets of extensions have been added to Gilt to make it significantly easier to create these user interfaces. The first set helps eliminate many of the call-back procedures which communicate to application programs. This was described in a previous paper[8]. The second set of extensions make it easier and faster for the designer to achieve the desired appearance for the user interface, and is described here.

In most toolkits, the widgets have many properties that the designer can set, such as the color, font, label string, orientation, size, the minimum and maximum values of a range, etc. Many widgets in the Motif widget set, for example, have nearly 50 different properties that can be set. Most interface builders, including Gilt, provide "property sheets" that allow the designer to specify the desired values (see Figure 2). However, it can be quite difficult and time consuming to find and set all of the appropriate properties. To show the magnitude of the problem, many applications contain over 2000 widgets, and the properties for each must be set in a consistent manner. A study has shown that achieving consistency in an interface is a frequently cited problem [9].

Another problem for interface designers is laying out the widgets in the window. When the designer places widgets with the mouse, they tend to be uneven and look sloppy. Therefore, most builders provide grids and alignment commands. However, these can be clumsy to use, and they do not insure that different dialog boxes will have a consistent alignment (for example, that the titles will always be centered at the top of the window).

To help solve these problems, Gilt introduces the notions of *Graphical Tabs* and *Graphical Styles*. These are based on the styles and tabs in text editors such as Microsoft Word. A "graphical tab" is simply a horizontal or vertical position in the graphics window to which objects can be aligned. A "graphical style" is a named set of properties and layout information, which can be applied to widgets. The designer can edit a widget so it has the desired properties, select it, and then define a named style based on it. The values of the properties and the position of the widgets will be associated with that style name. The style can then be applied to other widgets.

Furthermore, Gilt will try to automatically guess when to apply a style, so the designer does not have to. By guessing the appropriate properties and layout, Gilt makes the user interface design process significantly faster, since users can quickly and imprecisely place widgets, and the system will automatically neaten them. Since the inferencing is based on the styles the user has defined, rather than based on global, default rules, as in earlier systems like Peridot[5] and Druid[11], the inferred properties and positions are more likely to be correct.

A set of styles and tabs can be written to a file to form a *Graphical Style Sheet* which can be used to insure that multiple applications have a consistent appearance. If a style is edited, all widgets that are based on that style are automatically updated, so that the interfaces will continue to be consistent.

Gilt is a part of the Garnet system[7]. Garnet is a comprehensive user interface development environment containing many high-level tools, including Gilt, the Lapidary interactive design tool[6], and others. Garnet also contains a complete toolkit, which uses a prototype-instance object model, constraints, and a separation of the behaviors from the graphics. Gilt stands for the Garnet Interface Layout Tool, and it supports interfaces built using either the Garnet look-and-feel widget set or Motif look-and-feel widget set. (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.) If you are interested in getting Garnet, contact the second author. Gilt uses CommonLisp, but the ideas presented here are applicable to interface builder tools using conventional compiled languages.

RELATED WORK

Of course, there are a large number of commercial and research interface builders that lay out widgets, including the Prototyper for the Macintosh, UIMX for Motif, DialogEditor[1], the NeXT Interface Builder[14], Druid[11] and YUZU[12]. All of these have the same basic structure: there are two or more windows. One is the work window where the user interface is being created, and another is the widget window, sometimes called the "palette" containing the widgets that can be placed. (Typically, in addition to the standard interaction techniques like menus, radio buttons, check boxes, and scroll bars, there are also decorations like rectangles, lines and text labels that can be added to the picture. In this paper, these are all included when the word "widget" is used.) The designer selects a widget from the palette and places it in the work window using a mouse. Usually, the designer can change the position and size of widgets using the mouse, and edit other properties using dialog boxes or property sheets. The builders also provide many editing functions such as moving, copying, deleting, and aligning widgets, and reading and writing to a file.

Peridot[5] guessed alignment of graphical objects using global rules. Druid[11] applies a similar technique to widget alignment. When the designer adds a new widget in a window, Druid immediately tries to find other widgets in the window that the new widget might be aligned with. For example, when the designer creates a label below another existing label, Druid guesses that the new label and the

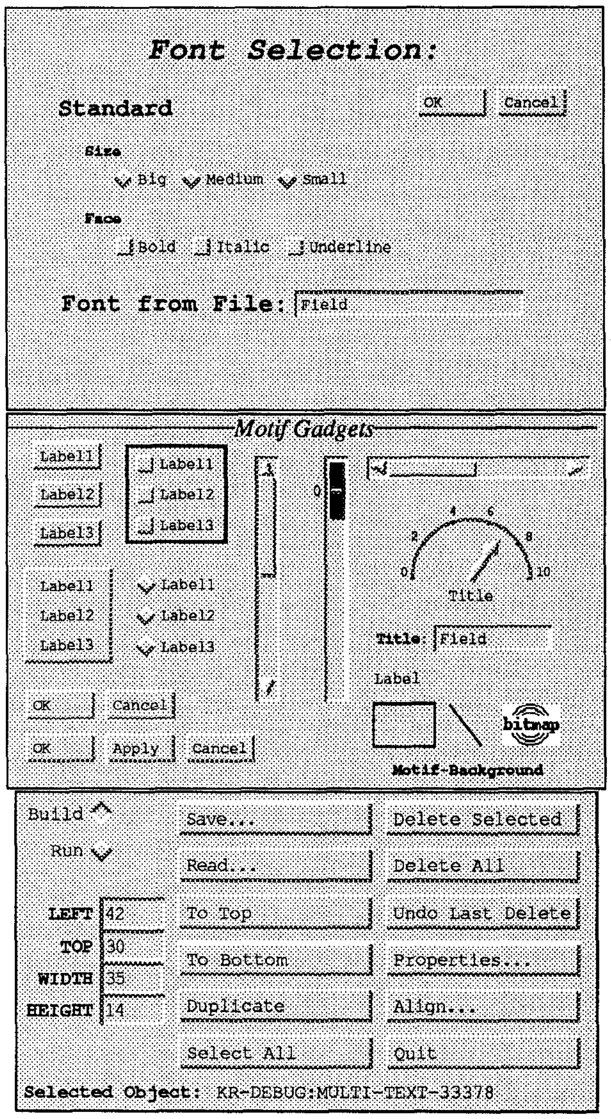


Figure 1: Gilt Main Windows
The top is the work window where a dialog box for a text editing application is being defined. The middle window is the palette of Garnet Motif gadgets that can be added to the work window. The bottom window is the main Gilt control panel containing the Gilt commands. The position and size of the selected widget is echoed in the text boxes at the left of this window.

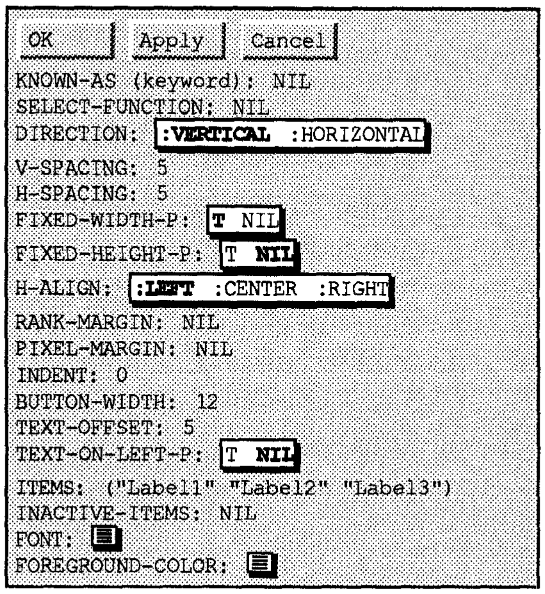
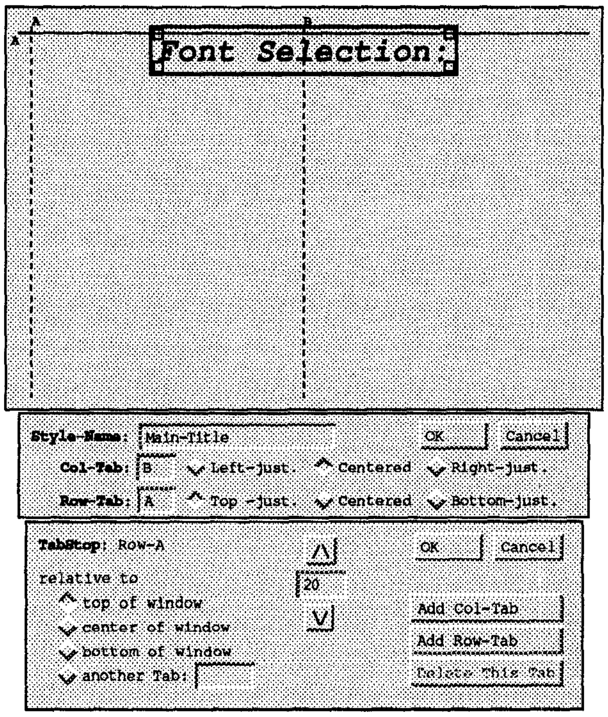

```
OK   Apply   Cancel
KNOWN-AS (keyword): NIL
SELECT-FUNCTION: NIL
DIRECTION: [✓ :VERTICAL] :HORIZONTAL
V-SPACING: 5
H-SPACING: 5
FIXED-WIDTH-P: [✓ T] NIL
FIXED-HEIGHT-P: T [✓ NIL]
H-ALIGN: [✓ :LEFT] :CENTER :RIGHT
RANK-MARGIN: NIL
PIXEL-MARGIN: NIL
INDENT: 0
BUTTON-WIDTH: 12
TEXT-OFFSET: 5
TEXT-ON-LEFT-P: T [✓ NIL]
ITEMS: ("Label1" "Label2" "Label3")
INACTIVE-ITEMS: NIL
FONT: ▤
FOREGROUND-COLOR: ▤
```

Figure 2: Gilt Property Sheet Window
The *Property Sheet* window for a set of check boxes. The designer can press with the cursor over any of the text fields, and type a new value. Pressing on the icon next to *Font* or *Foreground-Color* will bring up a sub-dialog box.

existing label should have the same left. It pops up a window so the designer can confirm the guess, and if the designer says yes, then Druid adjusts the objects automatically. However, Druid does not infer other properties of the objects, and the layout rules are hard-wired, rather than based on the user's preference, as in Gilt.

Many interface builders have provided interesting mechanisms for specifying the positions of widgets. For example, FormsVBT[2] and ibuild [13] use a "glue" model based on TeX. Glue has a varying stretch, and using the right kinds of glue between widgets causes the widgets to move appropriately when windows change size. In Lapidary[6], the designer can select two objects, and define arbitrary layout constraints between them. The most common constraints can be applied by using iconic menus. OPUS[3] shows the specified constraints as wires between the objects. We feel that the concept of tab stops will be more familiar to users and will be easier to use than these other approaches, while still providing most of the needed functionality. Also, no previous interactive builder has incorporated a notion of Graphical Styles, as used in Gilt.

The design for styles and tabs in Gilt is based on their use in text editors, in particular Microsoft Word for the Macintosh. This text editor allows the users to move a marker in a graphical ruler to set a tab stop, and if the TAB key is typed, the text cursor will move to the designated place. To define a style in Microsoft Word, the user formats some text in the



```
Style-Name: Main-Title                              OK   Cancel
   Col-Tab: B   Left-just.   [✓ Centered]   Right-just.
   Row-Tab: A   [✓ Top -just.]   Centered   Bottom-just.

TabStop: Row-A                  Λ        OK   Cancel
relative to                    20
   [✓ top of window]             V        Add Col-Tab
   center of window                      Add Row-Tab
   bottom of window                      Delete This Tab
   another Tab:
```

Figure 3: Style Editing Window and TabStop Window
Row-Tab A is selected in the work window (top), and is a horizontal tab that is 20 pixels from the top of the window, as shown by the *TabStop Editing* window at the bottom. The string "Font Selection:" is top-justified on Row-Tab A, and centered horizontally on Col-Tab B, which is centered in the window, so it will move if the window changes size. The *Style Editing* window (center) shows that the title is using the style *Main-Title* and Col-Tab B and Row-Tab A.

desired way, selects it, and then defines a new named style based on it. More general text styles are supported in [10].

GRAPHICAL TABS

An important graphic design principle is that widgets should be aligned evenly. This means that the edges or centers of the widgets should be the same, and that they should be evenly spaced. Furthermore, different dialog boxes should use the same alignments. For example, if in one dialog box a set of radio buttons is left justified under a title, and offset below it by 10 pixels, the same offset and alignment should generally be used in other dialog boxes.

Graphical tabs allow these kinds of relationships to be defined. A "graphical tab" is a horizontal or vertical position in a window. A horizontal tab position is specified relative to the top, bottom or center of a window. Similarly, a vertical tab is specified relative to the left, right or center. This allows the tabs to move appropriately if the window is resized. Just as with text editor tabs, the designer can specify whether the

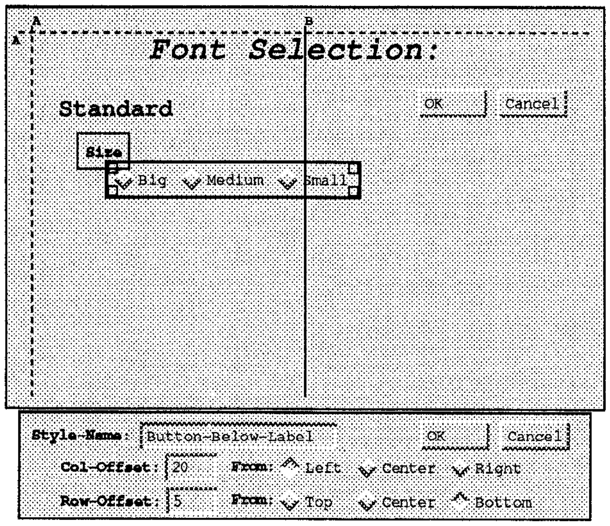


Figure 4: Style Editing Window for Relative Position
The position for the radio buttons is defined relative to the string "Size".

widgets will be left-justified, centered, or right-justified on the tab (or top-, centered, or bottom-justified for horizontal tabs). Since Garnet is implemented on X/11 which uses a pixel coordinate system, the offsets are specified in pixels. Gilt names the tabs with letters (although user-named tabs might be added in the future). Figure 3 shows a Gilt work window with a set of tabs visible. Whether the tabs are visible or not is controlled by a command.

New tab stops can be explicitly added by clicking on the "add tab" buttons in the *TabStop Editing* window shown at the bottom of Figure 3. Tabs can be selected by pointing on the label next to the line in the work window. The selected tab can then be deleted if no styles or widgets are using it. Tabs can be edited by entering new values into the tabstop editing window, or the tab stop labels in the work window act as handles and can be directly dragged with the mouse. When a tab is moved, all of the widgets defined using that tabs are also moved.

GRAPHICAL STYLES

A graphical style includes a set of widget properties, and optionally some position information as well. To create a new style, the designer modifies a widget to the desired appearance using the conventional property sheets, selects that widget, and then issues the *Define Style* command. The designer must then type a style name into the *Style Editing* window that will appear. Gilt compares the widget's current properties with the default values for that widget and copies all that are different. The widgets used to define the style are surrounded with a dark outline rectangle in the work window while the style is being defined or edited ("Font Selection:" in the top window of Figure 3).

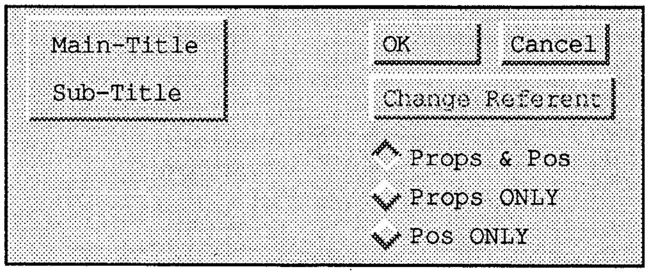


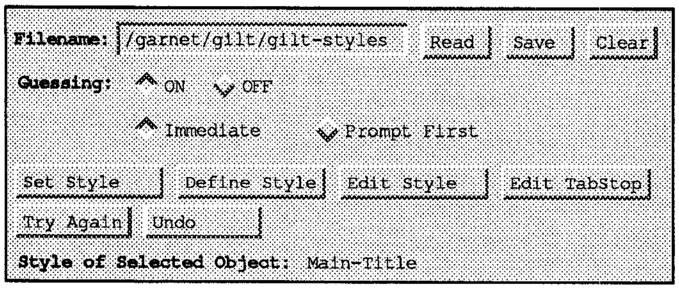
Figure 5: Set Style Window
This window allows designers to explicitly set a style. All the current defined styles are listed on the left, and the designer can choose one, and then specify whether the associated properties, position or both should be applied to the selected widget. If the selected style uses a relative position (Figure 4), then the *Change Referent* button is not grayed-out, and can be used to select the widget that the widget should be relative to.

Since all the widgets in the Garnet toolkit use the same names and values for similar properties, a style defined on one type of widget will often work on other types. For example, radio buttons, check boxes, and button sets all allow the designer to specify the orientation (horizontal or vertical) and fonts. In the top window of Figure 1, all the buttons have the same style properties. The types that styles are associated with include strings, buttons (which include radio buttons, check boxes and button sets), numeric sliders (which include both sliders and scroll bars), text input fields, etc.

Styles can also include position information. For example, a designer might specify that widgets with the *Main-Title* style should use a very large bold and italic font, and be centered at the top of the window. The position information for styles can either be with respect to a graphical tab stop, or relative to a previously created widget. For the first type, the appropriate tab name can be entered into the style editing window (see the center window of Figure 3). Either or both of the horizontal and vertical tab name fields can be blank, in which case no position information is recorded in that direction.

To specify that a style's position should be relative to another widget, the designer selects the referent widget after the style editing window is displayed. The style window will then change, as shown in Figure 4. When a style is relative, only the type of the referent widget is remembered. For example, in Figure 4, the style is defined as offset from any string. This will allow the *Button-Below-Label* style to be used relative to other strings, which can be in other parts of the window.

The *Set Style* window (see Figure 5) allows the designer to choose any of the defined styles, and also whether the position and properties aspects of the style should be applied. When setting the properties, Gilt checks each property associated with the style to see whether the widget accepts that property.



Filename: /garnet/gilt/gilt-styles

Guessing: ON OFF

Immediate Prompt First

Style of Selected Object: Main-Title

Figure 6: Style Control Window
This window allows styles to be read and written to a file, and style guessing to be turned on and off. Also, the style of the selected widget is always echoed at the bottom of the window.

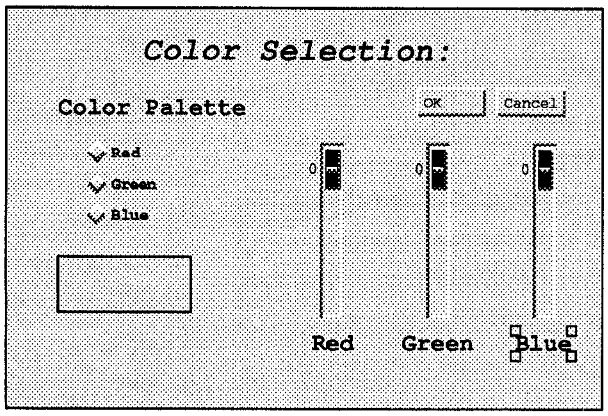


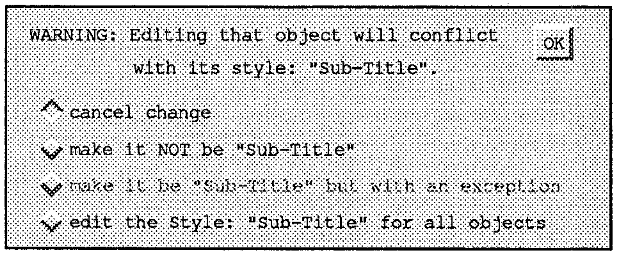
Figure 7: Inferring Styles

If not, then that property is ignored. For example, a style defined using radio buttons might have a value for the *Text-on-left-p* property, which determines which side the diamond is on. However, this is not relevant for push buttons (since their text is inside the button), so it would then be ignored. For styles with absolute positions, the widget simply moves to the correct tab stop. For relative positions, the user can specify the referent widget.

INFERRING STYLES

Although the styles mechanism as described above is already quite useful, Gilt goes further and tries to automatically determine when a particular style is appropriate. The *Style Control* window (Figure 6) provides three options: no inferencing of styles, styles applied immediately when they are inferred, or a prompt-first mode where the designer is asked if the style should be applied, as in Peridot and Druid. If the system usually infers the correct style, then the immediate mode will be the most efficient.

When inferencing is on, Gilt tries to infer a new style



WARNING: Editing that object will conflict
with its style: "Sub-Title".

cancel change

make it NOT be "Sub-Title"

make it be "Sub-Title" but with an exception

edit the Style: "Sub-Title" for all objects

Figure 8: Warning Window
This window pops up when the designer edits a widget that has a style attached to it.

whenever a widget is created or moved. The algorithm looks for styles that affect the same type as the widget, and checks how close the widget matches the style's position. For a style with a relative position, in order to find its possible referent widgets Gilt checks all the widgets of the appropriate type near the new widget. A list is created of all the styles that match, sorted according to the distance to the tab stops or the referent widgets. For example, in Figure 1 the main-title and the sub-titles use different styles with different fonts and positions, and Gilt can infer the appropriate style from the position when the designer places the new string.

Any inferencing system will sometimes guess wrong. Thus, it is important to provide appropriate feedback so the users are confident that they are in control and know what Gilt is doing. In immediate mode, the first style on the style list is immediately applied to the widget, and the name of the style is shown at the bottom of the style control window (Figure 6). The widget will also jump to the inferred position and change appearance. If the inferred style is not correct, the designer can hit the *Try Again* button (Figure 6), which will remove the guessed style and instead apply the next style in the sorted list. The *Undo* button can also be hit to remove the guessed style, and return the widget to its original position and properties. In prompt-first mode, the sorted list of all the inferred styles is presented in a window, with the most likely selected. The designer can select a different style, if necessary, and then hit *OK* or *Cancel.*

When a style is defined, it immediately becomes a candidate for inferencing. This is very useful when a number of widgets will all be created using the same style. In Figure 7, after the designer defines a style which centers the text label below the first scroll bar, when the second scroll bar and label are created, the label will automatically be centered. This highlights an advantage of the style approach over a rule-based approach as used in Druid and Peridot. Those systems might have put the label left-justified under the second scroll bar if it was placed closer to that alignment, but Gilt only matches against previously demonstrated styles, so it is more likely to guess the designer's intentions. This will also help achieve a consistent design.

EDITING STYLES

When a style is applied to a widget, either explicitly or inferred, Gilt sets up appropriate pointers and back pointers so that if the style is ever edited, all widgets using that style are immediately updated.

Styles can be edited in two ways. A property sheet can be displayed which shows the current values of the properties for the style, and this can be edited directly. This property sheet has the same format as the ones for the standard widgets (Figure 2). The position associated with the style can be edited using the appropriate dialog boxes (Figure 3 and 4).

Alternatively, the designer can edit the styles in the same way as they were created: by working on example widgets. Whenever a widget is edited that has already been defined to be of a particular style, Gilt pops up a dialog box asking if the edit should change the style itself (Figure 8). The other alternatives are to make the widget no longer belong to the style, or to cancel the change and return the widget to its appearance before the edit was attempted.

In the future, we plan to add the ability to have widgets use a particular style with exceptions, but this is a complex problem[4]. Some of the issues are whether to copy the attributes or retain the link to the original style, what to do to a style when the style it inherits from is changed, and whether to save the inheritance links in the style files, or write out all the style information to each file.

WRITING AND READING STYLES

A set of styles can be written to a file using the buttons in the style control window (Figure 6). This file can then act as a "Style Sheet." Whenever a new dialog box is being created, the style sheet file can first be read. Then, the appropriate styles can be inferred or explicitly applied to the widgets. This will help insure that the new dialog box is consistent with previous dialog boxes created for this or other applications.

When the work window is saved to a file, Gilt will optionally include the style information in that file. In this case, the file is self-contained. Alternatively, the file can simply contain a pointer to the appropriate style file. Then, whenever the window is used by applications or read back into Gilt, the style file will be re-read, so any subsequent edits to the styles will be reflected. However, this can cause the window to look ugly (for example, if the style for a set of radio buttons changes from horizontal to vertical, the buttons are likely to overlap other widgets). Therefore, a version number is kept in the style file, so at least a warning can be issued when an old window is opened with an edited style file.

EVALUATION

As a small, informal experiment to see how quickly users could create interfaces, using grapical styles and tabs, four subjects were given two tasks. Each task has two similar dialog boxes. For the first box in each task (i.e.DBox1 and DBox3), the properties for all widgets were set using the

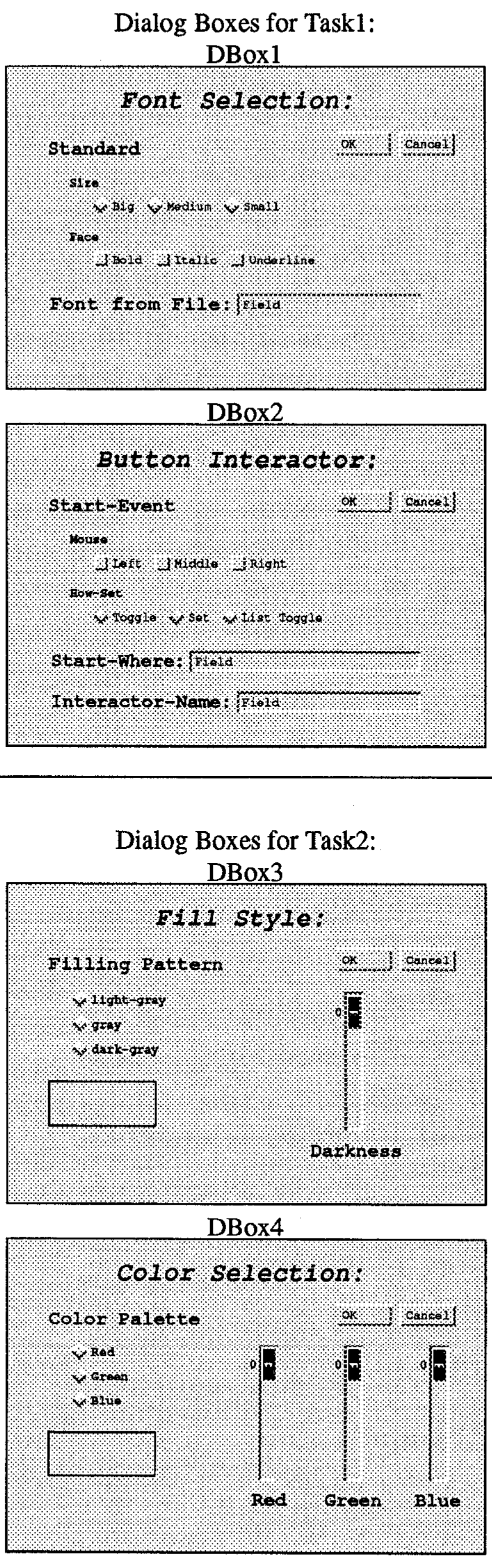


Figure 9: Dialog Boxes for Task1 and Task2

property sheets. Their positions were determined using tabs. Then, several styles were defined using these properties and positions. For the second box in each task (i.e.DBox2 and DBox4), the properties and positions for all widgets can be inferred automatically, using the styles defined in the first box (Table 1, Figure 9). The same four dialog boxes are created without any styles by a Gilt expert. The results were used to compare with above results (Table 2 and 3).

Table 1: Task Description

	Task1		Task2	
	DBox1	DBox2	DBox3	DBox4
Widgets	8	9	7	11
Defined TabStops	3	0	1	0
Defined Styles	6	0	4	0
Guessed Widgets	2	9	3	11

From Table 2 and 3, it is clear that the dialog box, where all widgets are inferred, is created in less than half the time for dialog boxes without styles: a 0.45 ratio for DBox2 and a 0.42 ratio for DBox4. In guessing the styles for users on DBox2 and DBox4, Gilt guessed correctly almost all the time. The over-heads for defining styles and tabs are small: a 1.73 ratio for DBox1 and a 1.28 ratio for DBox3. Note, however, that the longer time for DBox1 is mostly due to the learning time since this was the first time using Gilt and styles for almost all subjects. In addition, novices can learn Gilt styles and tabs quickly, because, in DBox1, they needed a 1.73 ratio, but, in DBox3, they needed only a 1.28 ratio.

The verbal protocols for these subjects indicated that they felt that Gilt style guessing was useful and comfortable. Two subjects said that the "Try Again" button is very good. Subject A, who took this test twice, using styles and without any styles, felt that defining relative styles was very useful, because the conventional layout mechanism did not support "offset" among widgets, so he often had to calculate "left + width + offset" values for the referent widget, in order to determine the left hand position for the new widget. He indicated that graphical tabs were good for aligning some widgets at particular fixed lines. However, all subjects claimed that they had to think about style names, whenever defining any styles. They indicated that it was difficult to give good names for all styles. Also, they said that sometimes they couldn't remember whether this name had already been used or not. Thus, we plan to prepare a default style name in the style editing window.

STATUS AND FUTURE WORK

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

In the future, we plan to investigate unifying tabs with the

Table 2: Task1 Results

[sec]	DBox1	DBox2	DBox1+2
Style: Subject A	420	160	580
Style: Subject B	1000	200	1200
Style: Subject C	910	300	1210
Style: Subject D	1060	270	1330
Style: Average	847	233	1080
No Style: Subject A	490	510	1000
Ratio	1.73	0.45	1.08

Table 3: Task2 Results

[sec]	DBox3	DBox4	DBox3+4
Style: Subject A	250	110	360
Style: Subject B	400	300	700
Style: Subject C	380	180	560
Style: Subject D	500	180	680
Style: Average	383	193	575
No Style: Subject A	300	460	760
Ratio	1.28	0.42	0.76

relative styles. It seems like there should be a convenient way to define "relative tabs" that will achieve the desired results. As discussed above, we would also like to investigate exceptions to the styles. There might be a way to copy just some values from one style into another, and ways to read just a few styles from a file. Further work is needed on ways for the system to automatically generalize styles, so that, for example, the font property or color defined on a radio button will be applied to a circular gauge, even though they have different types.

CONCLUSIONS

The *Graphical Styles* mechanism described in this paper can help designers more quickly create user interfaces, because many of the properties and alignments can be applied with a single specification, or even inferred automatically. In addition, the styles can help insure consistency across multiple dialog boxes in an application, and even across multiple applications, since *Style Sheets* can be developed and re-used. The *Graphical Tab* mechanism seems to be an easier-to-understand and easier-to-edit mechanism than other layout approaches. Finally, in addition to being useful for user interface builders, such as Gilt and YUZU, we feel that the graphical styles and graphical tab mechanisms would be useful for a wide range of graphical editors, including drawing programs and CAD/CAM.

ACKNOWLEDGEMENTS

Andrew Mickish helped to implement the features described in this article. Brad Vander Zanden and other members of

Garnet project provided useful advice and help with the design and implementation. For help with this paper, we want to thank Dave Kosbie, Richard McDaniel, Andy Mickish, Francesmary Modugno, Bernita Myers, Brad Vander Zanden, Tomonari Kanba, Hiroshi Yamada, Kin-ichi Hisamatsu, Kyoji Kawagoe, the YUZU development members and the reviewers.

This research was partially sponsored by NEC Corporation, and partially 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.

REFERENCES

[1] Luca Cardelli. Building User Interfaces by Direct manipulation, In *Proceedings of UIST' 88*, Alberta, Canada, 1988, pp.152-166.

[2] Gideon Avrahami, Kenneth P.Brooks, and Marc H.Brown. A Two-View Approach To Constructing User Interfaces, In *Proceedings of SIGGRAPH'89*, Boston, 1989, pp.137-146.

[3] Scott E.Hudson and Shamim P.Mohamed. Interactive Specification of Flexible Interface Displays, *ACM Transactions on Information Systems 8,3*, 1990, pp.269-288.

[4] Jeff Johnson and Richard J.Beach. Styles in Document Editing Systems, *IEEE Computer 21,1*, 1988, pp.32-43.

[5] Brad A.Myers. Creating User Interfaces by Demonstration, *Academic Press*, 1988.

[6] Brad A.Myers, Brad Vander Zanden, and Roger B.Dannenberg. Creating Graphical Interactive Application Objects by Demonstration, In *Proceedings of UIST'89*, Williamsburgh, 1989, pp.95-104.

[7] Brad A.Myers, Dario A.Giuse, Roger B.Dannenberg, Brad Vander Zanden, David S.Kosbie, Edward Pervin, Andrew Mickish, and Philippe Marchal. Garnet: Comprehensive Support for Graphical Highly-Interactive User Interfaces, *IEEE Computer 23,11*, 1990, pp.71-85.

[8] Brad A.Myers. Separating Application Code from Toolkits: Eliminating the Spaghetti of Call-Backs, In *Proceedings of UIST'91*, Hilton Head, 1991, pp.211-220.

[9] Brad A.Myers and Mary Beth Rosson. Survey on User Interface Programming, In *Proceedings of CHI'92*, Monterey, 1992, pp.195-202.

[10] Brad A.Myers. Text Formatting by Demonstration, In *Proceedings of CHI'90*, New Orleans, 1991, pp.251-256.

[11] Gurminder Singh, Chun Hong Kok, and Teng Ye Ngan. Druid: A System for Demonstrational Rapid User Interface Development, In *Proceedings of UIST' 90*, Snowbird, 1990, pp.167-177.

[12] Takahiro Sugiyama *et al.*. CANAE User Interface Builder: YUZU (In Japanese), In *Proceedings of the 45th National Convention of Information Processing Society of Japan*, Tokushima, 1992, 5Q-3.

[13] John M.Vlissides and Steven Tang. A Unidraw-Based User Interface Builder, In *Proceedings of UIST'91*, Hilton Head, 1991, pp.201-210.

[14] Bruce F.Webster. The NeXT Book, *Addison-Wesley Publishing*, 1989.