Automatic, Look-and-Feel Independent Dialog Creation for Graphical User Interfaces

Brad Vander Zanden
Brad A. Myers

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
bvz@cs.cmu.edu

ABSTRACT

Jade is a new interactive tool that automatically creates graphical input dialogs such as dialog boxes and menus. Application programmers write a textual specification of a dialog's contents. This specification contains absolutely no graphical information and thus is look-and-feel independent. The graphic artist uses a direct manipulation graphical editor to define the rules, graphical objects, interaction techniques, and decorations that will govern the dialog's look-and-feel, and stores the results in a look and feel database. Jade combines the application programmer's specification with the look-and-feel database to automatically generate a graphical dialog. If necessary, the graphic artist can then edit the resulting dialog using a graphical editor and these edits will be remembered by Jade, even if the original textual specification is modified. By eliminating all graphical references from the dialog's content specification, Jade requires only the absolutely minimum specification from the application programmer. This also allows a dialog box or menu's look and feel to be rapidly and effortlessly changed by simply switching look and feel databases. Finally, Jade permits complex inter-field relationships to be specified in a simple manner.

KEYWORDS: Automatic Dialog Layout, Look-And-Feel Independence, Direct Manipulation, Graphical Specification

INTRODUCTION

Jade is a new tool that automatically creates and lays out graphical input dialogs, such as menus, palettes, buttons, and dialog boxes. It does this by combining a textual specification of the dialog's contents, written by an application programmer, with look-and-feel databases prepared by a graphic artist or style expert. The dialog can then be modified by a graphic artist using a direct manipulation, graphical editor, and Jade will maintain these edits, even if the original textual specification changes.

The specification that the application programmer writes is completely look-and-feel independent. Indeed, there are no references in the specification to graphics of any kind. Jade obtains all the graphic information it needs from the style files prepared by the graphic artists. Thus the look-and-feel of a dialog can be effortlessly changed from a Garnet style to an OpenLook style by simply switching look and feel databases (see Figure 1). No changes to the textual specification are required. By eliminating all graphical references from a dialog's specification, Jade requires only the absolutely minimum specification from the application programmer: the dialog's contents and the desired interaction techniques (e.g., menu, single-choice or multiple-choice buttons, number-in-a-range, etc).

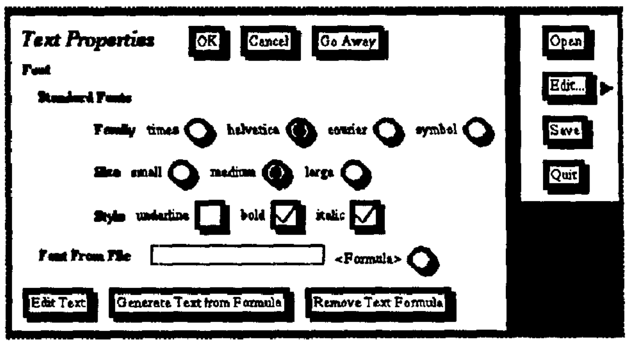
Jade also makes it possible to specify complex interrelationships in a compact, simple form. For example, the programmer can specify that certain sections of the dialog should be disabled (e.g., grayed out) unless another portion of the dialog has been selected. This feature can be used to gray out the buttons that control the parameters in a move or grow operation unless the corresponding operation is selected. It is also easy to specify how the behavior of various groups of buttons are related. For example, to control a move operation, the application programmer may present the user with three buttons—a "left" button that specifies that the x coordinate of the object should be changed, a "top" button that specifies that the y coordinate of the object should be changed, and a "formula" button that specifies that a formula will decide which coordinates should be changed. The programmer can easily specify that the "left" and "top" buttons are mutually exclusive with the "formula" button, but that the "left" and "top" buttons can be simultaneously selected.

The graphic artist creates the look-and-feel for a dialog using a direct manipulation, graphical editor called Lapidary [5]. To create a look-and-feel, the designer works with an example dialog that has been generated by Jade. The style expert can change the look-and-feel's graphics by creating custom graphical objects and associating them with interaction techniques that Jade recognizes, such as menu or number in a range behaviors. Or if the interaction technique does not exist, the designer can make use of a number of tools that are provided with the graphical editor to create the desired behavior.

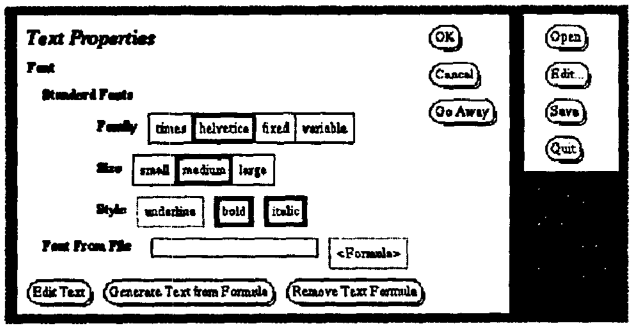
The graphic artist specifies the look-and-feel's layout by either selecting positioning rules from a dialog box or by demonstrating the rules graphically. For example, a group of buttons that controls the dialog, such as "OK" and "Cancel", can be positioned by selecting a rule from the rules dialog box, or by moving the group to an appropriate position on the screen and asking Jade to infer the desired rule. The graphic artist is also free to add decorations to the look and feel, such as enclosing menus in rectangles or placing the word "or" between mutually exclusive items. Thus the graphic artist can visually experiment with different combinations of layout rules, decorations, and objects to obtain the desired look-and-feel.

Finally, the graphic artist is allowed to edit any dialog generated by Jade, using the same graphical editor that creates the looks and feels. Jade will remember these changes and ensure that they are applied to the dialog, even if the original textual specification is modified.

Jade, which stands for (a Judgement-based Automatic Dialog Editor), is one of the tools that is being developed as part of the Garnet project at Carnegie Mellon University [7]. Garnet seeks to provide a comprehensive environment for creating direct manipulation, graphical interfaces. Garnet currently provides an object and constraint package, a graphics package, and an interactors package that handles input devices at a high level [6]. Also under active development is a graphical editor, called Lapidary [5], that allows *all* pictorial aspects plus most behavioral aspects of user interfaces to be specified graphically. Jade currently generates all the dialog boxes used in Lapidary, such as the one in Figure 2. All of Garnet, including Jade, is implemented in Common Lisp



(a)



(b)

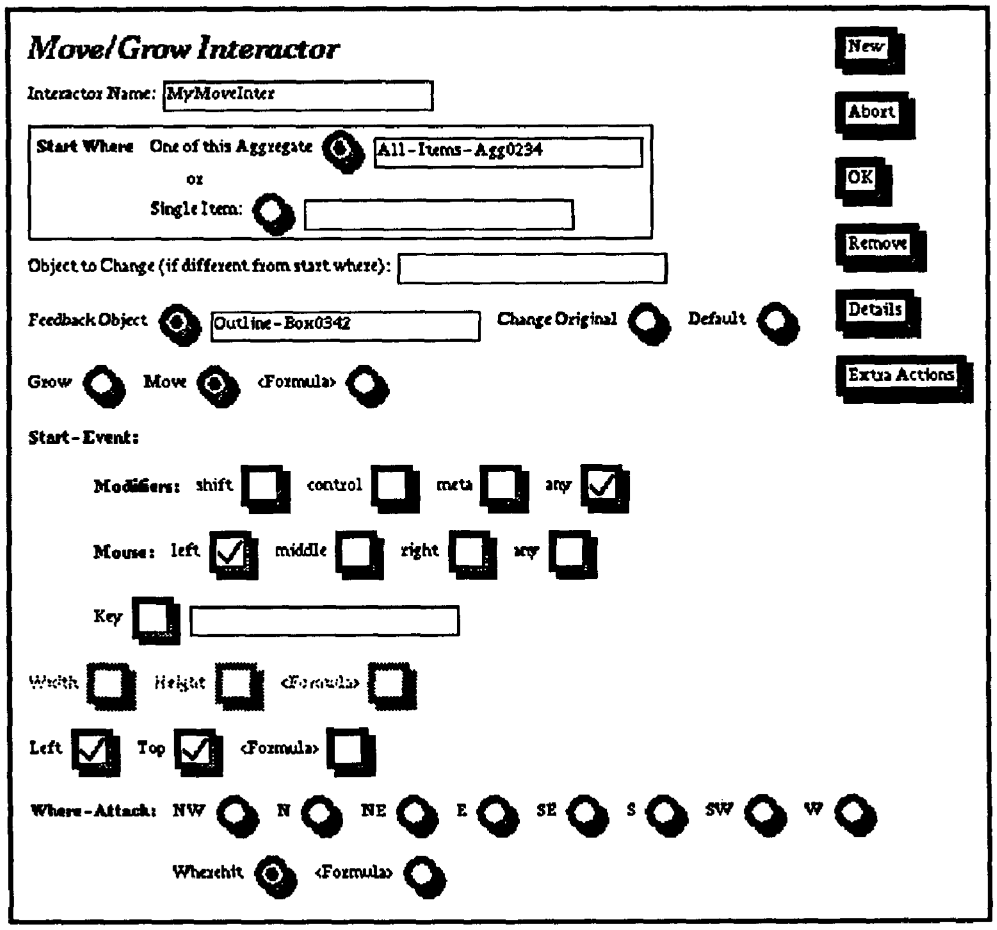
Figure 1.

A Garnet style (a) and an OpenLook style (b) for specifying the properties of a text object and controlling an application. The "Edit" menu item has an associated submenu whose presence is indicated in a look-and-feel specific manner. The Garnet and OpenLook dialogs were generated from the same textual specification.

on top of the X Window System. It is currently available under Allegro, Lucid, and CMU Common Lisps and it runs on both IBM RT and SUN workstations. Garnet is designed to be portable so it should not be difficult to compile it under other versions of common lisps or to make it run on other machines. The lower levels of Garnet (*not* including Jade and Lapidary) are available for free under license. Contact the authors for more information.

RELATED WORK

Interface builders such as NeXT's, the Macintosh Prototyper by Smethers Barnes, and DialogEditor [3] allow a designer to select objects from a predefined set and position them by hand. However, this can be a slow, inaccurate process. Jade improves this by automatically positioning and aligning objects, while still providing the graphic artist with a direct manipulation editor that



Move/Grow Interactor

Interactor Name: MyMoveInter

Start Where One of this Aggregate: All-Items-Agg0234
or
Single Item:

Object to Change (if different from start where):

Feedback Object: Outline-Box0342 Change Original Default

Grow Move <Formula>

Start-Event:

Modifiers: shift control meta any ✓

Mouse: left ✓ middle right any

Key

Width Height <Formula>

Left ✓ Top ✓ <Formula>

Where-Attach: NW N NE E SE S SW W

WhereHit <Formula>

Figure 2.

A dialog box created by Jade that is used by Lapidary to specify a move or grow behavior. The application programmer has instructed Jade to gray out the parameters associated with the move or grow interactor unless the button corresponding to that interactor is chosen, and the graphic artist has enhanced the Jade-created dialog box by placing a rectangle around the Start-Where group and placing the word "or" between that group's items.

can change the resulting layout. Jade also permits the graphic artist to quickly create new objects that can be immediately used in the dialog.

Tools such as Mikey [8], the Interactive Transaction System [1, 10], Scope [2], and Chisel [9] automatically generate dialog boxes from textual specifications. Mikey generates Macintosh dialogs from the type descriptors found in a Pascal program and Scope generates dialogs based on the Athena toolkit from C++ programs. The Interactive Transaction System generates dialog boxes from a specification of the dialog's content and a set of style rules created by a style expert. Chisel automatically lays out the presentation component of an interface based on a specification of the dialog's contents, a set of guidelines established by a designer, and a list of user preferences. Jade differs from these systems in a number of ways: by providing a direct manipulation editor that allows a graphic artist to modify the dialog once it has been created and retaining these edits if the original specification is modified, by making it easy to create and add new objects to the dialog, and by allowing the graphic artist to create new looks and feels via a direct manipulation, graphical editor. Jade also differs from Mikey and Scope in that it permits dialog specifications to be used with multiple looks and feels.

A HIGH-LEVEL OVERVIEW OF JADE

Figure 3 presents a high-level overview of Jade's architecture. Jade takes a textual specification prepared by an application programmer, an optional file of exceptions, and a look-and-feel graphics file and a look-and-feel rules file prepared by a graphic artist. As output it generates a graphical layout for a dialog. The look-and-feel graphics file contains graphical objects and decorations and is created using Lapidary; the look-and-feel rules file contains positioning information and is created using the Jade rule editor. The Jade rule editor obtains its rules from a master rule database, which can be augmented by rules that are demonstrated to Jade by the graphic artist and by interaction techniques that the graphic artist demonstrates or that the application programmer writes using Garnet's interactors package.

An application programmer writes a specification by giving the labels for the dialog items and the names of the interaction techniques that these items represent. Labels are provided as lists. For example, the list `("underline" "bold" "italic")` might denote the options for the style of a font. Since several groups of options may be combined to produce one final result, label lists can be nested to obtain subgroups. For example, a font is determined by its family, its size, and its style. Thus the portion of the dialog that determines the font might be written as:

```
("Standard-Fonts"
  ("Family" ("times" "helvetica"
             "courier" "symbol"))
  ("Size" ("small" "medium" "large"))
  ("Style" ("roman" "bold" "italic")
           (:behavior :multiple-choice)))
```

Jade will indent the *Family*, *Size*, and *Style* groups so that users can visually recognize that they are related to *Font*, as shown in Figure 1.

After listing the contents of a group, the application programmer can provide a number of slot-value pairs that define the group's interaction technique. The `:behavior` slot selects one of the following seven built-in interaction techniques:

- Single-Choice: Allows a user to select only one item at a time from a group of items.
- Multiple-Choice: Allows a user to select multiple items from a group of items.
- Text: Allows the user to input a string of text.
- Single-Choice-With-Text: Allows the application programmer to associate a type-in text field with the single-choice interaction technique so that the user can input a string of text. This interaction technique is used in the "Start Where" portion of Figure 2.
- Multiple-Choice-With-Text: Allows the application programmer to associate a type-in text field with the multiple-choice interaction technique so that the user can input a string of text.
- Command: Allows the user to select items from a menu.
- Number-in-a-Range: Allows the application programmer to specify a slider-like interaction technique that will permit the user to input a number in a specified interval.

Jade automatically selects the appropriate graphical object based on the value of the `:behavior` slot and information in the look-and-feel graphics file. If these parameters are omitted, Jade will provide the default interaction technique, single-choice. `:single-choice` and `:multiple-choice` may be optionally followed by a list of items, which may be the labels of dialog objects and groups, or lists themselves. Every item on this list will exhibit the single-choice or multiple-choice behavior, depending on the prefixing keyword. For example, suppose that in the *family* group in Figure 1, the bold and italic options can be chosen simultaneously, but neither can be selected if the underline option is chosen. This behavior can be easily written as

```
(:behavior :single-choice
  ("underline"
   (:multiple-choice "bold" "italic")))
```

Jade will also automatically generate code to enable or disable groups of items based on the value of the `:enable` slot. Disabled items might be "grayed out" (the graphic artist determines the actual appearance of disabled items). The value of the `:enable` slot will generally depend on whether some other item in the dialog is selected. Therefore, the application programmer will write a "formula" that expresses a relationship between the value of the `:enable` slot and the "enabling" item. A formula can be any arbitrary Lisp expression. The constraint layer of Garnet provides a special command called *gv* that can be used to access the value of a slot in Garnet objects (gv stands for get value). The Garnet constraint system automatically reevaluates a formula when the value of the referenced slot changes.

In writing formulas it is also important to know that Jade uses a tree structure to represent a dialog, with the groups as interior nodes and the items as leaves. Jade labels these nodes with the names given to groups and items. The application programmer can access these nodes through slots that are stored in the nodes' parents. In this case, the names of the slots are the names of the nodes they point to. For example, the *Standard Fonts* group in Figure 1 will have a slot named `:standard-fonts` in its parent that contains a pointer to the *Standard Fonts* group. Thus, to access the "times" button in Figure 1 from a sibling of the *Font* family, the expression `(gv :self :parent :standard-fonts :family :times)` suffices.

As an example of how formulas might be used in the `:enable` slot, consider the move/grow operation represented by Figure 2. The group represented by `Width`, `Height`, and `Formula` should be enabled when the grow button is selected. To accomplish this, the application programmer would write the formula `(gv :self :parent :move-grow-group :grow :selected)`. This formula instructs Jade to go to the

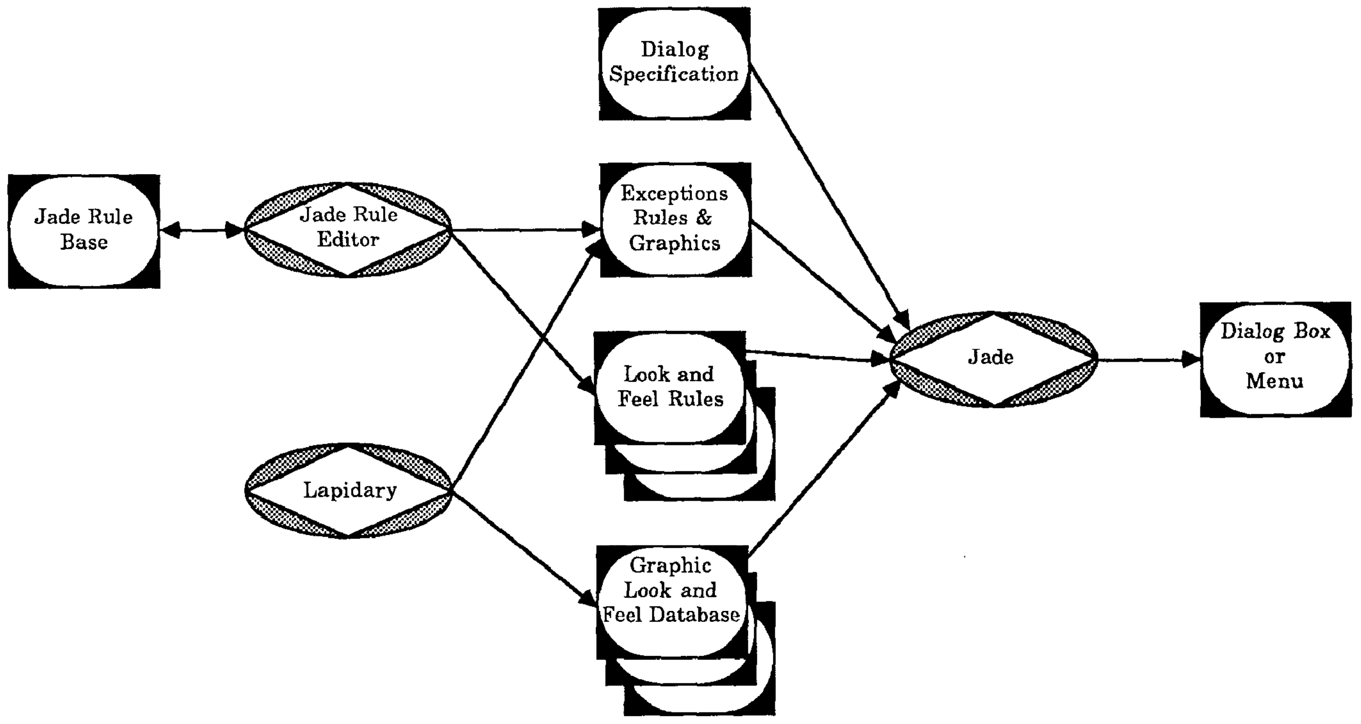


Figure 3.

Architecture of the Jade system. Processes are denoted by diamonds inscribed in ovals and input and output are denoted by roundtangles inscribed in rectangles. Jade's final output, a dialog box or menu, is part of the run-time environment.

parent of the *Width-Height-Formula* group, then to the *Move-Grow* group, and then to the grow button where it will check if the grow button is selected. If so, it will enable the Width-Height-Formula group, otherwise it will disable it (represented by the grayed out group in Figure 2).

Yet another slot that the application programmer may provide is the `:stop-action` slot, which gives the name of a function that should be called when the user selects one of the items in a group. This function is called with the name of the selected object. It may also access any other object in the dialog using the tree structure outlined earlier. As an alternative to using stop action functions, the application programmer may place formulas that depend on the items in the dialog in application objects. The objects will then be automatically updated by Garnet's constraint system when the user makes a new selection.

Constraints and stop-actions allow the application programmer to manage the flow-of-control in a larger application context with many dialogs. For example, the application programmer can provide a stop action that makes another dialog visible when a button is hit. Or the application programmer can place a constraint in the `:enable` slot of a button that causes it to be dim unless a button in another dialog is selected.

Finally, a specification may have a special *stop-group* slot that specifies the functions that control the dialog, such as "OK" and "Cancel". Jade treats the stop-group as a command behavior. To assist the programmer in implementing the stop functions, Jade maintains a list of the groups whose selections have been modified and passes this list to the stop action. Jade also positions the stop-group specially, using different layout rules than the rules used to lay out the other dialog groups.

The specification in Figure 4 generates the dialog box and menu shown in Figure 1. The sublist (`"copy" "cut" "paste" "delete"`) causes a submenu for "edit" to be created, which will appear after "Edit" has been selected. In contrast, the sublist (`"times" "helvetica" "courier" "symbol"`) causes a string of buttons to be laid out, prefixed by the non-selectable label `Family`. The selectability of labels is determined by the interaction technique they are associated with.

```
(create-dialog
    ((("Font"
       (("Standard-Fonts"
          (("Family" ("times" "helvetica" "courier" "symbol"))
           ("Size" ("small" "medium" "large"))
           ("Style" ("roman" "bold" "italic")
                   (:behavior :single-choice
                      ("underline" (:multiple-choice "bold" "italic")))))
        ("Font From File"
          (:behavior :text))
        "Formula")))
      (("Edit Text" "Generate Text from Formula" "Remove Text Formula")
         (:behavior :command)
         (:stop-action text-handler)))
     (:stop-group (("OK" "Cancel" "Go Away")
                   (:stop-action 'font-stop-action)))))

(create-dialog
      ((("open" ("edit" ("copy" "cut" "paste" "delete")) "save" "quit")
          (:behavior :command)
          (:stop-action control-menu-handler))))
```

Figure 4.

Textual specification of the dialog box shown in Figure 1.

AUTOMATICALLY GENERATING DIALOG BOXES

In order to automatically lay out a dialog, Jade uses several heuristics based on the particular look-and-feel, such as where to position the stop group, whether to start each group on a new line, how to position subgroups, and how to position items within a group. It also needs to know what types of graphical objects to use. To help it make these decisions, Jade consults a rule base that is maintained in a look-and-feel file. The rules in this look-and-feel file are in turn derived from a master rule base that Jade maintains. A graphic artist can use a graphical editor, as described in the next section, to create new rules or the application programmer can write new interaction techniques using the Garnet interactors package.

There are rules for placing objects, determining when to break a line, for specifying horizontal and vertical offsets, and for determining what type of font should be used. These rules are of the form:

(rule-name rule-body parameters)

The parameters are optional and may include things like objects and offsets that help the placement rules position objects with respect to another object. Sometimes the rule body is as simple as a number indicating how many pixels should vertically separate two groups of items or how many pixels a subgroup should be indented. Other times the rule body consists of a list of slot-names, such as `:left` or `:top`, that parameterize an object, and definitions for these slots, such as a formula indicating that the top of an object should be a certain number of pixels below the bottom of another object. An example rule might be

```
(at-right-rule
 ((:left (formula (+ (gv obj :right) offset)))
  (:top (formula (gv obj :top))))
 obj offset)
```

This rule positions an object to the right of the reference object. The formula for `:left` finds the right margin of the reference object and adds to this a number of pixels equal to the offset. The formula for `:top` finds the top margin of the reference object. Jade places these formulas and values directly into the dialog groups and objects. The Garnet constraint system then evaluates the formulas and the results dictate the layout of the dialog.

The look-and-feel file associates rules with layout parameters that Jade uses to position objects in the dialog. These bindings may also restrict the scope of a rule. For example, a rule may apply to the layout of a specific group of items, such as the *Family* group in the text properties dialog box, or to an interaction technique, such as single-choice items, or to the whole dialog. These bindings have the form:

(parameter-name rule-name applies-to)

For example, if *at-right-rule* applies only to the alignment of single-choice items, the binding would be:

(:alignment at-right-rule :single-choice)

When multiple bindings apply, Jade chooses the most restrictive one. Thus, when laying out a single-choice group, Jade will prefer the *at-right-rule* binding over a binding that applies to the whole dialog.

It is assumed that the graphic artist who creates these rules and bindings will not have much, if any, programming experience. Therefore, as discussed in the next section, Jade provides a special editor that allows the graphics artist to demonstrate rules and bindings. Jade stores the demonstrated rules and bindings in the format just described. In the event that a rule or binding cannot be expressed in this editor, the graphic artist can enlist the services of a programmer, who can write the rules or bindings using the above format, and add them to the appropriate database (for simplicity, the rules and binding syntax mirrors the syntax used in Garnet).

ADJUSTING DIALOGS, RULES, AND STYLES

Once Jade has laid out a dialog, a graphic artist may want to modify it. For example, the designer may want to add decorations, such as rectangles around groups, or modify some aspect of the layout, such as the way items within a group are positioned with respect to one another. Jade needs to remember these changes so that it can apply them, even if the application programmer edits the original specification. The designer may also want some of these changes to modify the look-and-feel so that other dialogs can be modified in the same manner. Finally, the designer may want to change the graphical objects that represent the various interaction techniques or create new interaction techniques.

All of these adjustments can be accomplished using the Lapidary editor. To modify the rules governing the layout of a dialog, the graphic artist can use a rule dialog box that shows the rules in the master rule base, or the designer can demonstrate the rule. Related rules are grouped together by the rule dialog box and the one that is currently used in the dialog is highlighted. For example, the rules that can affect how items in a group are positioned will be displayed together. When the designer selects a new rule, the Jade generated dialog is immediately updated to reflect the change.

The graphic artist can also change the rules in a direct manipulation manner by repositioning dialog objects, such as moving a stop-group to the bottom of the dialog and aligning it horizontally. Jade will then use Peridot-style inferencing [4] to guess which rule from the master rule base should be applied. If none of the rules seems to apply, Jade will ask the designer if it should create a new rule to cover this situation. The designer can then give this rule a name and Jade will record the rule in its rule database. The designer is then free to incorporate this rule into the current look-and-feel, other look-and-feels, or only into this dialog.

To add decorations to a dialog, the graphic artist draws the objects in Lapidary and then positions them using Lapidary's constraint menus. The constraints refer to the group's or item's name, so Jade can remember the exceptions by saving the decorations, the constraints, and the group's or item's name. As long as the names do not change when the original specification is edited, Jade will be able to position the decorations correctly, even if the size or position of the items changes. If a name changes or disappears, Jade will ask the designer whether the decoration should be deleted, or whether it should be keyed to a new name.

The designer can also change the graphics that represent the various Jade interaction techniques by creating new graphical objects using Lapidary and then linking them to the appropriate Jade interaction technique. Jade will immediately insert the new graphical object into the dialog using the look-and-feel rules that the designer has defined. If the desired interaction technique is not recognized by Jade, such as a two-choice behavior that allows the user to select exactly two items, the graphic artist can either demonstrate the interaction technique using Lapidary, or ask the application programmer to code the interaction technique using Garnet's interactors package. This interaction technique can then be added to Jade's rule base and incorporated into other look-and-feels.

The changes the designer makes to the layout rules, decorations, graphical objects, and interaction techniques can affect multiple levels in Jade. Graphical changes can be stored in an exceptions file, in which case they will only apply to the dialog being edited, or they can be stored in a look-and-feel graphics file, in which case they will apply to all dialogs created with that look-and-feel. Rule changes can be stored in an exceptions file, a look-and-feel rules file, or the master rule base. If they are stored in the master rule base, they will affect all look-and-feels.

Within a dialog, the designer can control whether a rule change has a local or global effect through the selection of dialog groups. If the designer preselects one or more groups, the rule change will only apply to these groups, otherwise the rule change will apply to the entire dialog. Alternatively, the designer may want a rule change to apply to a specific kind of group, for example, any group that consists entirely of multiple-choice text objects. In this case, the designer can select a representative group, apply the rule to it, and then ask Jade to generalize it to all groups of that type.

CURRENT STATUS AND FUTURE WORK

Jade is currently used to create the dialog boxes in Lapidary and was used to generate all the dialog figures in this paper. The present implementation of Jade consults look-and-feel databases to determine which rules and graphical objects it should use. Rules can also be changed locally so that the modifications only apply to a particular group.

The master rule base has not been implemented so it is not currently possible to extend Jade by adding new rules or interaction techniques. The Jade rule editor has also not yet been implemented so Jade cannot yet infer rules or have rules demonstrated to it. However, it is possible to modify rules textually by placing the names of the appropriate rules in a look-and-feel file. Similarly, graphical objects can be created using Lapidary, and then linked textually with Jade interaction techniques.

A possible future extension is to use Jade to generate a special editor for creating a Jade specification. Of course this editor would use Jade created menus and dialog boxes. For example, the `:behavior`, `:stop-action` and `:enable` slots could be placed in a dialog box along with fields for a group's label and the graphical objects' labels. A mechanism for defining subgroups could also be established by creating multiple copies of the dialog box, one for each subgroup. The advantage of this approach is that it would be easier for the designer to create a syntactically correct specification. The disadvantage is that it would not be as fast as simply typing in the specification.

CONCLUSIONS

Jade presents a new technique for rapidly creating graphical dialogs with the same look-and-feel, allowing a graphic artist to modify the resulting dialog, and remembering these modifications even if the original specification is edited. By providing look-and-feel databases, Jade allows the dialog specification itself to be completely look-and-feel independent. Thus an application programmer can create dialogs by simply listing the contents of the dialog, and a graphic artist or style expert can create the rules, graphics, decorations, and interaction techniques that govern the look-and-feel of these dialogs. In addition to allowing the style expert to alter the dialogs once they have been created, Jade will permit the style expert to create new rules, graphics, interaction techniques, and decorations using a graphical, direct manipulation interface. Thus Jade will be extendable and will permit the graphic artist to visually experiment with different looks and feels when designing an application's graphical user interface.

ACKNOWLEDGEMENTS

We are grateful for the help we received from the other Garnet project members in designing Jade: Dario Giuse, Roger B. Dannenberg, David Kosbie, Philippe Marchal, and Ed Pervin.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, Amendment 20, under contract F33615-87-C-1499, monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), Wright-Patterson AFB, Ohio 45433-6543. 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 Defense Advanced Research Projects Agency or the US Government.

REFERENCES

1. William E. Bennett, Stephen J. Boies, John D. Gould, Sharon L. Greene, and Charles F. Wiecha. Transformations on a Dialog Tree: Rule-Based Mapping of Content to Style. Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology, Williamsburg, VA, Nov., 1989, pp. 67-75.

2. Clifford M. Beshers and Steven Feiner. Scope: Automated Generation of Graphical Interfaces. Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology, Williamsburg, VA, Nov., 1989, pp. 76-85.

3. Luca Cardelli. Building User Interfaces by Direct Manipulation. Proceedings of the ACM SIGGRAPH Symposium on User Interface Software, Banff, Alberta, Canada, Oct., 1988, pp. 152-166.

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

5. Brad A. Myers, Brad Vander Zanden, and Roger B. Dannenberg. Creating Graphical Objects by Demonstration. Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology, Williamsburg, VA, Nov., 1989, pp. 95-104.

6. Brad A. Myers, Dario Giuse, Roger B. Dannenberg, Brad Vander Zanden, David Kosbie, Philippe Marchal, Ed Pervin, and John A. Kolojejchick. The Garnet Toolkit Reference Manuals: Support for Highly-Interactive, Graphical User Interfaces in Lisp. Tech. Rept. CMU-CS-89-196, Carnegie Mellon University Computer Science Department, Nov., 1989.

7. Brad A. Myers. An Object-Oriented, Constraint-Based, User Interface Development Environment for X in Common Lisp. 4th Annual X Technical Conference, Boston, MA, Jan., 1990. To appear.

8. 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.

9. Gurminder Singh and Mark Green. Chisel: A System for Creating Highly Interactive Screen Layouts. Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology, Williamsburg, VA, Nov., 1989, pp. 86-94.

10. Charles Wiecha, William Bennet, Stephen Boies, and John Gould. Generating user interfaces to highly interactive applications. Human Factors in Computing Systems, Proceedings SIGCHI'89, Austin, TX, April, 1989, pp. 277-282.