The Lapidary Graphical Interface Design Tool

Brad Vander Zanden

Computer Science Department
University of Tennessee
Knoxville, TN 37996
bvz@cs.utk.edu

Brad A. Myers

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

The Lapidary interface design tool allows the graphics and run-time behaviors that go *inside* an application window, as well as the graphics and behaviors that go around the application window, such as menus and scroll bars, to be specified pictorially. In particular, Lapidary allows the designer to draw pictures of application-specific graphical objects that the end user will manipulate (such as boxes and arrows, or elements of a list), the feedback that shows which objects are selected (such as small boxes on the sides and corners of an object), and the dynamic feedback objects (such as hair-line boxes to show where an object is being dragged). The run-time behavior of all these objects can be specified in a straightforward way using constraints and dialog boxes that allow the designer to provide abstract descriptions of the interactive response to the input devices. Lapidary also generalizes from the specific example pictures to allow the graphics and behaviors to be specified by demonstration. Finally, Lapidary supports the construction and use of ''widgets'' (sometimes called interaction techniques or gadgets) such as menus, scroll bars, buttons and icons. Thus Lapidary allows *all* pictorial aspects of an application to be specified graphically.

CR Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques—*User Interfaces*; I.3.6 [**Computer Graphics**]: Methodology and Techniques.

General Terms: Human Factors.

Additional Key Words and Phrases: User Interface Management Systems, Interaction, Object-Oriented Design, Direct Manipulation, Interaction Techniques, Programming by Example.

1 Introduction

Lapidary is an interactive design tool that allows designers to pictorially specify all graphical aspects of an application, including both the objects that control an application, such as menus, scroll bars, and buttons, and the application-specific graphics themselves. The application-specific graphics may include the feedback objects that indicate selections, such as selection handles, the feedback objects that track the mouse, such as hair-line boxes that indicate where an object is being dragged, and the components of the application itself, such as boxes and arrows. In addition, Lapidary allows the run-time behavior of objects to be specified. Constraints can be defined graphically and by demonstration to describe how the objects relate to one another (e.g., that arrows stay attached to boxes), and the objects' responses to the mouse and other input devices can be specified or demonstrated using abstract *interactors* that embody particular kinds of interactive behaviors.

A number of commercial and research toolkits provide limited capabilities to create user interfaces. They typically provide developers with a palette of graphical items, such as menus, buttons, and scrollbars, and allow copies of these items to be arranged in various graphical layouts (Figure 1). However, these graphical items are application-independent in the sense that they can be reused across various applications. The tools provide no support for defining the application-specific graphics, laying out these graphics, or controlling the way a user interacts with these graphics. For example, in the boxes and arrows editor shown in Figure 2, the object palette in the upper left corner allows the user to select which type of object should be drawn next, the delete-quit menu allows arrows or boxes to be deleted from the drawing and allows the user to exit the application, and the drawing window on the right allows the user to draw labeled boxes and to connect them using arrows. A typical graphical design toolkit would allow a developer to create the delete-quit menu and lay it out in a work window (Figure 1). It would provide no support for 1) creating and laying out the object palette; 2) creating the application's graphical objects, such as the labeled boxes and arrows; 3) defining the layout of the output, such as keeping the endpoints of the arrows centered within the boxes they connect; and 4) defining the ways in which the user can interact with the output, such as moving or resizing the labeled boxes, or changing the labels. In contrast, Lapidary supports all four of these tasks.

It is important to note that the time required to accomplish each of these four tasks far outweighs the time required to create and position the delete-quit menu. Indeed, it has been estimated that the amount of time required to develop the application specific graphics and behaviors is about two orders of magnitude greater than the time required to create the controlling gadgets that surround an application window [27]. However, as the above example illustrates, today's interface builders do not support defining the application-specific graphics. Lapidary does so, and in a fashion that allows the designer to create interfaces in the most direct, intuitive and straightforward manner possible: by graphical drawing and demonstration. There are a number of reasons why using such an interactive editor is preferable to using a programming language. First, since graphical interfaces are pictorial, it is easier to create them by drawing the appropriate pictures rather than trying to describe the pictures using a cumbersome, textual programming language. Second, using an interactive tool is *much* faster than programming, both because of the simplicity of creating pictures and because the designer can immediately see and change what the final application will look like. Directly manipulating the graphics is faster than repeatedly modifying and running a program that generates the graphics. Finally, a non-programmer graphics artist can directly create an interface using an interactive tool instead of having to rely on an applications programmer.

Lapidary is part of the Garnet user interface development environment [17] and is one of the interactive tools that provide a visual programming environment for Garnet. Garnet helps programmers develop graphical interfaces by providing an object-oriented programming environment, a graphical object system, a constraint system, and a behavioral model called ‘‘interactors’’ for handling input. Lapidary provides access to the graphical object system through a direct manipulation drawing editor, supports simple constraints using two iconic menus that graphically depict the most commonly used alignment constraints, allows more complicated constraints to be created using a spreadsheet editor called C32 [19], and provides dialog boxes that allow users to specify the behavior of objects by entering parameters for various interactors. Lapidary also supports the demonstration of certain types of behaviors through ‘‘before’’ and ‘‘after’’ inferencing. Since the user creates example pictures and behaviors, Lapidary will also perform inferencing in order to generalize these pictures and behaviors into abstract entities that can be instanced at run-time by the application.

Garnet is implemented in CommonLisp on top of the X window system. As a result, both Garnet and Lapidary are portable to various workstations and operating systems. For example, Garnet runs on CMU, Lucid, Allegro, and AKCL Common Lisps and has been used on IBM, Sun, DecStation, TI, and HP workstations. A ‘‘Lapidary’’ is a workman who cuts, polishes and engraves precious stones, and here is a Lisp-Based Assistant for Prototyping Interface Designs Allowing Remarkable Yield. Lapidary has been released and can be acquired by anonymous FTP as part of Garnet. For information on how to do this, contact `garnet@cs.cmu.edu`. A complete reference manual for Lapidary is also available [21].

2 Related Work

Lapidary can be classified under the broad heading of User Interface Management Systems [15]. It provides capabilities that are found in interface builders, widget builders, presentation builders, behavior builders, and constraint builders. However, each of these types of tools allows only a limited set of graphics or behaviors to be specified. Lapidary goes beyond these tools in that it allows arbitrary application graphics and their behaviors to be pictorially specified.

2.1 Interface Builders

There have been a large number of systems that allow users to select items from a pre-defined library of widgets, and position them on the screen. Some of these, including Menulay [2], Trillium [6], DialogEditor [3], vu [28], Druid [29], Gilt [20], and the NeXT Interface Builder, provide a graphical editor that allows the position and size of each widget to be specified in a direct manipulation manner using a mouse. Often, these systems allow a few limited properties to be changed using a dialog box. Lapidary also provides the ability to create and align pre-defined widgets using its drawing and constraint editors.

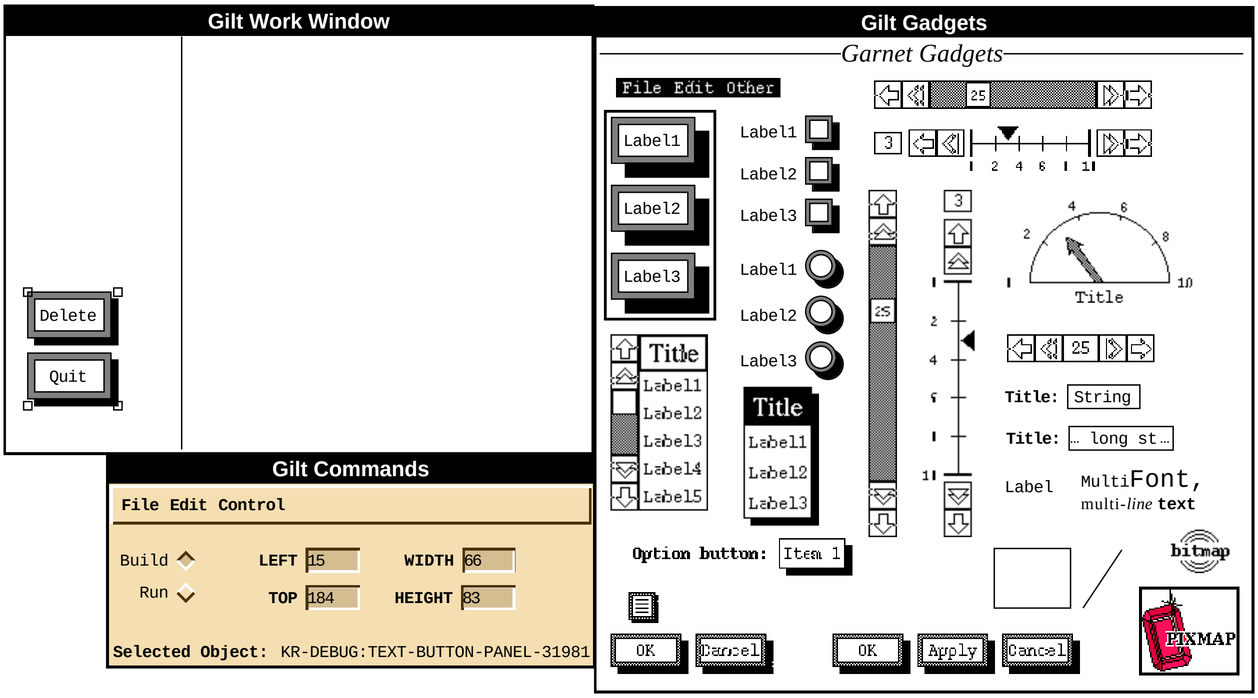


Figure 1: Current graphical toolkits provide a palette of graphical items which can be copied and placed in a work window. Here, the Gilt interface builder, which is part of the Garnet user interface development toolkit, is being used to create the delete-quit menu for the boxes and arrows application shown in Figure 2.

2.2 Widget Builders

The Peridot system [13, 14] supports creating the widgets themselves using direct manipulation. It provides a drawing editor that allows a user to create widgets with an arbitrary look-and-feel. It also provides an inferencing facility that helps align objects and attempts to guess the behaviors the widgets should exhibit. Lapidary also allows widgets to be created from scratch with an arbitrary look-and-feel. However, Peridot contains a great deal of knowledge about various interaction techniques, such as menus and sliders, and thus is able to perform much more sophisticated inferencing of widget behaviors. Lapidary is more general than Peridot since it can also handle application-specific graphics, so it does not have the knowledge that would allow this style of inferencing. Instead it allows behaviors to be created through dialog boxes and a limited form of ‘‘before’’ and ‘‘after’’ demonstration (see section 7.3).

2.3 Presentation Builders

Most interface builders primarily support the construction of relatively static panels or dialog boxes where the objects in the panels are menus, buttons or text fill-in slots that do not move around. A number of systems support the creation of more dynamic presentations. For example, XY-Wins [5] allows users to draw mobile objects, such as elements of visual programming languages. However,

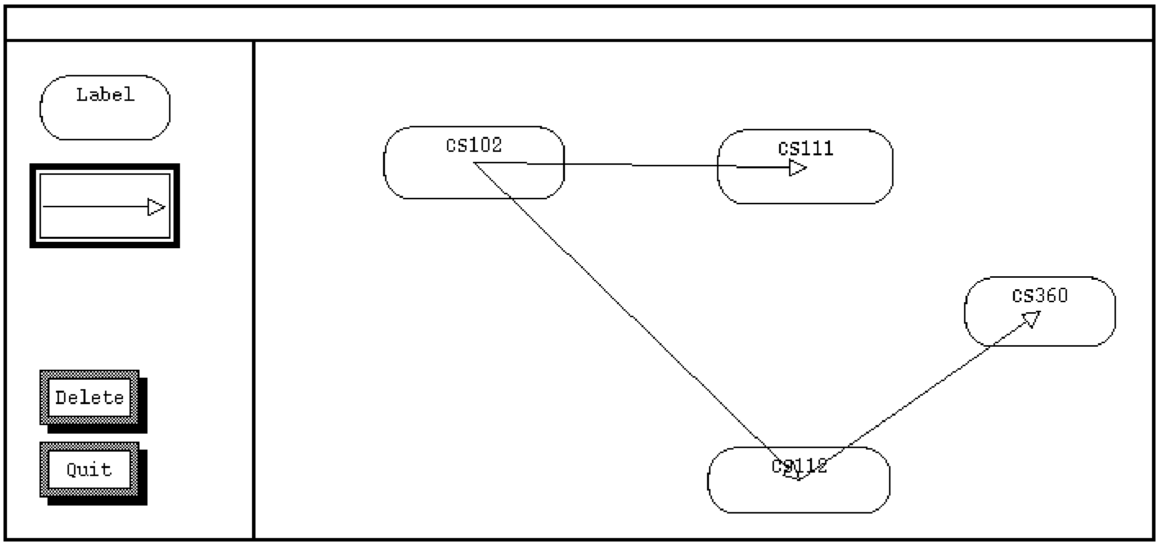


Figure 2: A boxes and arrows editor. Existing graphical toolkits can be used to create the delete-quit menu but cannot be used to create 1) the application graphics, such as the labeled box and arrow, or 2) the application graphics behavior, such as the arrow's endpoints staying attached to the centers of boxes. In contrast, Lapidary allows all these graphics and behaviors to be created in a direct manipulation manner.

these systems still allow only limited properties to change at run time (such as position). Other systems, such as EDGE [25] support more dynamic behavior by limiting the applications to a certain form, such as those that use graphs to display their data. Ibuild [31] goes beyond most interface builders in that it allows arbitrary toolkit elements or graphical components to be laid out and behaviors to be attached to these elements and components. For example, Ibuild can be used to create the presentation of a text editor, complete with scroll bars and pull-down menus. It can also be used to define pre-defined relationships between various objects. For example, if the user brings up the relate tool and selects a scroll bar and the text editor, Ibuild will infer that the scroll bar should scroll the text in the text editor. However, Ibuild does not support the specification of the dynamic presentation of objects (e.g., an arrow should stay connected to a box), or the graphical creation of new behaviors. All behaviors must be pre-defined by textually coding them. Lapidary improves on the capabilities of existing presentation builders by allowing all aspects of the dynamic presentation of an application to be pictorially specified, including the properties that should change at run-time and the behaviors that objects should exhibit.

2.4 Behavior Builders

A number of demonstrational systems have been developed that assist in the specification of the dynamic behavior of objects in an application. Examples include systems for creating user interfaces [29, 32, 23], constructing drawings and animating them [11, 12], and creating HyperCard programs [4]. These systems record the user's actions on example objects and then play them back at run-time when an appropriate event is received. In some cases they attempt to generalize the sequences of actions to provide more abstract behaviors. For example, Demo [32] will create linear equations that relate elements of an event (e.g., rotating a dial on a gauge by 30 degrees) with elements of a response (e.g., updating a text box with the new value of the gauge). MetaMouse [11, 12] and Eager [4] try to detect repetitive actions and generalize the sequence of actions so that they are performed on all objects in a group. Marquise [23] concentrates on inferring the overall behavior of an application from demonstrations.

Lapidary differs from these behavior builders in that it provides a predefined set of behaviors, such as selection from a set of choices or entering a text string, and allows the user to customize the behaviors by setting various parameters. It also allows some types of behavior to be demonstrated using ‘‘before’’ and ‘‘after’’ presentations (see Section 7.3 for details) and constraints. Thus Lapidary requires the user to explicitly specify behavior rather than demonstrating it through a sequence of actions. One advantage that these behavior builders have over Lapidary is that they can specify more of the ‘‘semantic’’ behavior of an application, such as creating new objects. On the other hand, by providing behaviors that are in effect pre-recorded sequences of actions and allowing the user to customize them using parameters, Lapidary allows many behaviors to be specified more rapidly and with less tedium. Lapidary also provides more flexibility in editing behaviors, since behaviors can be modified by simply changing a parameter, as opposed to redemonstrating an entire sequence. Finally, Lapidary-provided behaviors are more general in that they can be reused in many parts of an interface by simply changing parameters. Behaviors that are defined by recording a set of actions typically cannot be easily modified to meet a slightly changed situation—most often they would have to be replayed to a certain point, changed, then replayed some more, and so on.

2.5 Constraint Builders

Much of the behavior in Lapidary is specified using constraints on objects. A constraint is a relationship among objects that is defined once and then maintained automatically by the system, even when the objects change. Like Peridot [13] and Apogee [7], Lapidary uses one-way constraints, which means that a property of one object (e.g., the `LEFT` of object `A`) can depend on another object (`B.LEFT`), but the reverse is not implied (if `B.LEFT` changes, `A.LEFT` is changed automatically, but if `A.LEFT` is changed, `B.LEFT` is not changed by the system).[1] Other constraint systems, such as ThingLab [1], have two-way constraints, but widespread experience with these systems is limited and there are issues associated with performance and user control over the constraint solving process that still have not been completely resolved.

A number of interface and presentation builders allow the static layout of applications to be specified using constraints, including DialogEditor [3], vu [28], Druid [29], and Apogee [7]. A variant of constraints, called glue, is used in Ibuild [31]. Rockit [9] and Artkit [8] provide capabilities for inferencing constraints that can be utilized by interface builders. Lapidary differs from these systems in

[1]Actually, the constraints in Lapidary are slightly more general, because cycles *are* allowed. Thus `A.LEFT` can depend on `B.LEFT` and `B.LEFT` can depend on `A.LEFT`. When either changes, the other is updated. What is disallowed, however, is any situation where a property has more than one constraint that calculates its value.

that it allows constraints to define the dynamic presentation of objects. It also differs from Rockit and Artkit in that it requires explicit application of constraints.

3 Overview of Lapidary

Lapidary can be used in every part of the interface building process:

- Creating widgets: In this paper we use the term ''widget'' to mean an interaction technique for controlling an application or for gathering input from a user. Although the designer will frequently be able to use pre-defined widgets, for example, from a Motif or Garnet toolkit, there are often application-specific widgets that must be specially created for a particular application. The operations palette in Figure 2 is a good example of such a widget.

- Creating application objects: Lapidary provides a direct manipulation drawing editor that allows designers to either create objects from scratch or use pre-existing objects. Lapidary provides access to both Garnet's primitive object types, such as rectangles and text, as well as its composition mechanism, which allows more complicated objects to be composed from simpler objects. This mechanism allows a user to create arbitrary application graphics. For example, Lapidary can be used to create prototypes for the labeled boxes and arrows used in the boxes-and-arrows editor in Figure 2.

- Specifying the static layout of objects: Designers can position widgets and static application graphics using the mouse. For example, the layout of the components of the operations palette and the positioning of the operations palette and delete/quit menu can be specified in Lapidary.

- Specifying the dynamic connections among objects: Designers can use Lapidary's constraint menus to define how objects should react when they are manipulated by a user. For example, Lapidary can be used to attach constraints to an example arrow that connects the endpoints of an arrow to the centers of two example boxes. The example arrow will be converted to a prototype arrow by Lapidary, so that at run-time, all instances of this arrow contain constraints that connect the instances to the centers of boxes. Lapidary will make the boxes be parameters, not constants, and allow the instances to determine at run-time which boxes they should be attached to. If a box is moved by a user, the constraint solver will automatically reposition all arrows attached to it so that the arrows stay attached to the center of the box. Similarly, constraints can be used to attach a feedback box to the mouse so that it follows the mouse as the mouse moves, or to cause a box that encloses a label to grow or shrink as the label grows or shrinks.

- Specifying interactive behavior: The designer can specify how objects should respond to input events through demonstration or by entering parameters into behavior-defining dialog boxes. Garnet divides behaviors into a number of categories such as object creation, text entry, making a selection from a set of choices, and moving or resizing an object. Each of these behaviors needs to have some information in order to operate properly. For example, the choice behavior needs to know which objects may be chosen, how it should highlight an object when it is chosen, and how many items in the set can be chosen at the same time. Lapidary provides dialog boxes that allow a designer to enter this information.

  As an alternative to the dialog boxes, the designer may choose to directly demonstrate a particular behavior. To do this, the designer tells Lapidary to remember the state of an object and then edits the object to indicate the desired outcome of the behavior. Lapidary then compares the ''before'' and ''after'' states and makes the appropriate set of inferences to implement the behavior.

4 Creating Objects

Lapidary provides a conventional, direct-manipulation drawing editor for creating and modifying objects. As shown in Figure 3, this editor includes a drawing window, a shapes palette, constraint menus for aligning objects, and property menus for changing the properties of objects. The shape menu provides the usual range of graphical primitives, so objects can be created from scratch. For example, in Figure 3 a list element is being created from rectangles, strings, and arrows. Alternatively, objects can be created by copying them or adding them from predefined sets of ‘‘prototypes.’’ For example, to bring in pre-defined widgets from Garnet’s own widget set or the Garnet-provided Motif widget set, the user can load instances of them, and then position them using Lapidary.

Certain properties of an object, such as position or size, can be edited in a direct manipulation manner. Other properties, such as line-styles, filling-styles, and fonts can be changed by bringing up the appropriate property menu (e.g., Figure 3.b). Finally, infrequently changed properties of an object, such as the curvature of the rounded corners of a roundtangle, can be modified using C32, Garnet’s spreadsheet editor, which shows all the properties of an object [19].

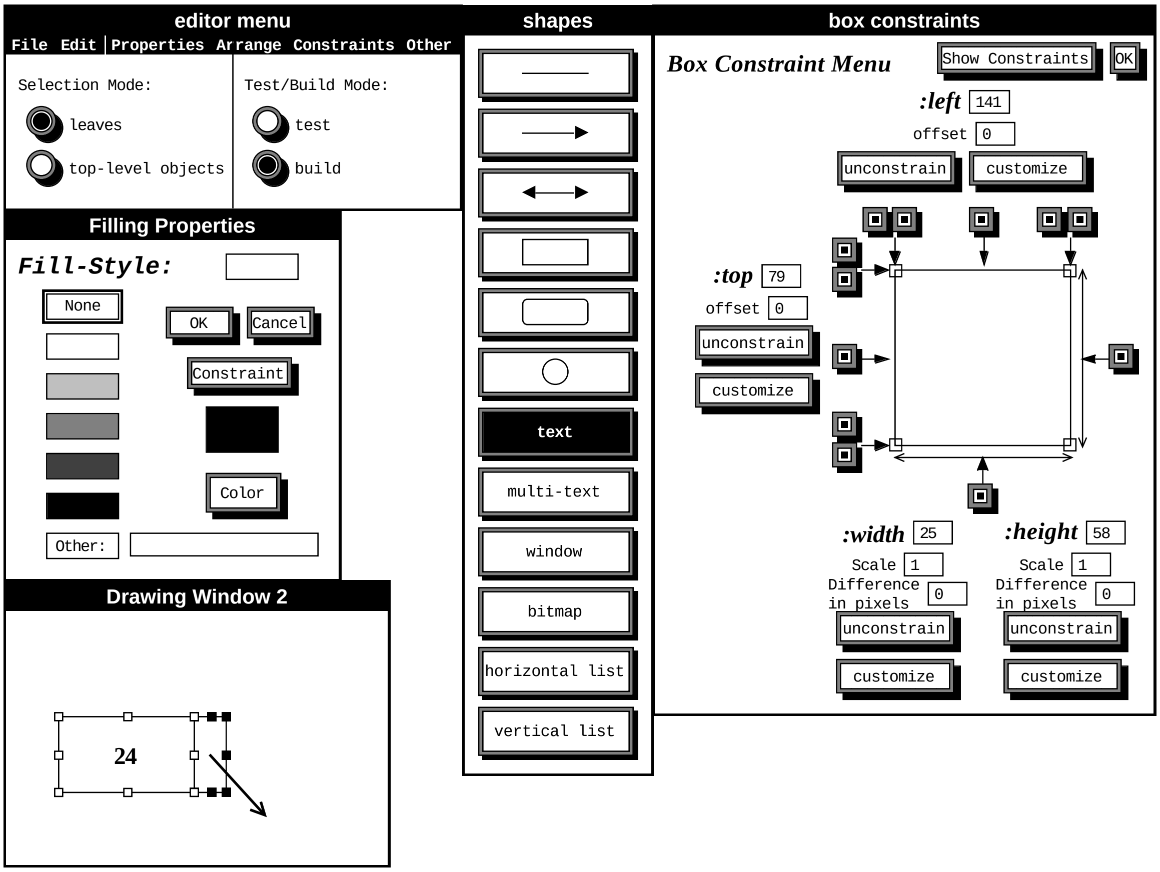
Lapidary allows objects to be aligned using constraints as described in the next section. However, there are occasions when the designer would like objects to be organized into higher-level entities that align objects, such as lists. Lapidary provides a framework for doing this through its list mechanism. A designer can select a prototypical item and then sweep out a list. Lapidary will create instances of this prototypical item and place them in the list. For example, once the designer has completed constructing the list element in Figure 3, the designer could select the list element, select the `horizontal list` option from the shapes palette, and sweep out a list containing instances of the list element. The designer could then customize the format of the list by changing various properties that control the alignment, spacing, and justification of objects [21]. At run-time, the application could control the number of objects in the list and the placement of objects in the list using various Garnet commands such as `add-item` and `remove-item`.

4.1 Implementation

Lapidary objects are represented using KR, a frame-based knowledge representation system. In KR, there is no distinction between instances and classes; any instance can serve as a ‘‘prototype’’ for other instances [10]. All data and methods are stored in ‘‘slots’’ (sometimes called fields or instance variables). Data and method slots[2] that are not overridden by a particular instance inherit their values from their prototypes. An instance can have any number of additional slots as well.

Lapidary allows prototypes to be constructed by example: any slot of an instance can be changed, and then that instance can be used as a prototype for new objects. Changes to a slot of the prototype are reflected in all instances that do not override that slot. This makes it easier to edit the look and feel of an entire interface by just editing the prototypes from which the interface was created. For example, if the

[2]There is no distinction between data and method slots in KR. Any slot can hold any type of value, and in CommonLisp, a function is just a type of value.



(d) (e)

Figure 3: The various Lapidary windows that support the creation of objects. The drawing window (c) contains an object that will be used to represent the elements of a list. The constraint menu (e) has been used to align the label within the ‘‘data’’ rectangle, the arrow within the ‘‘pointer’’ rectangle, and the pointer rectangle to the right of the ‘‘data’’ rectangle. Window (a) contains the main Lapidary commands, window (d) contains the shapes that can be created in Lapidary, and the dialog box (b) represents one of Lapidary’s property menus.

font in a labeled box prototype were changed, all labeled box instances created from it would change.

Example objects created in Lapidary can serve as prototypes in run-time applications, since they can be cloned by a run-time application using Garnet’s built-in instancing mechanism. Thus once a designer has created an object in Lapidary, the object can be saved and loaded into any application. Garnet provides a saving mechanism that allows Lapidary to write objects to disk in a format that can be read back later into either Lapidary or an application [22]. The output file is simply a header with some Lapidary information (the header is Lisp-executable) and a list of Garnet commands and Lisp code that will recreate the objects. Thus it is similar to the code that a programmer would create using a a conventional text editor. As a result, the programmer can read the Lapidary-generated code, modify it,

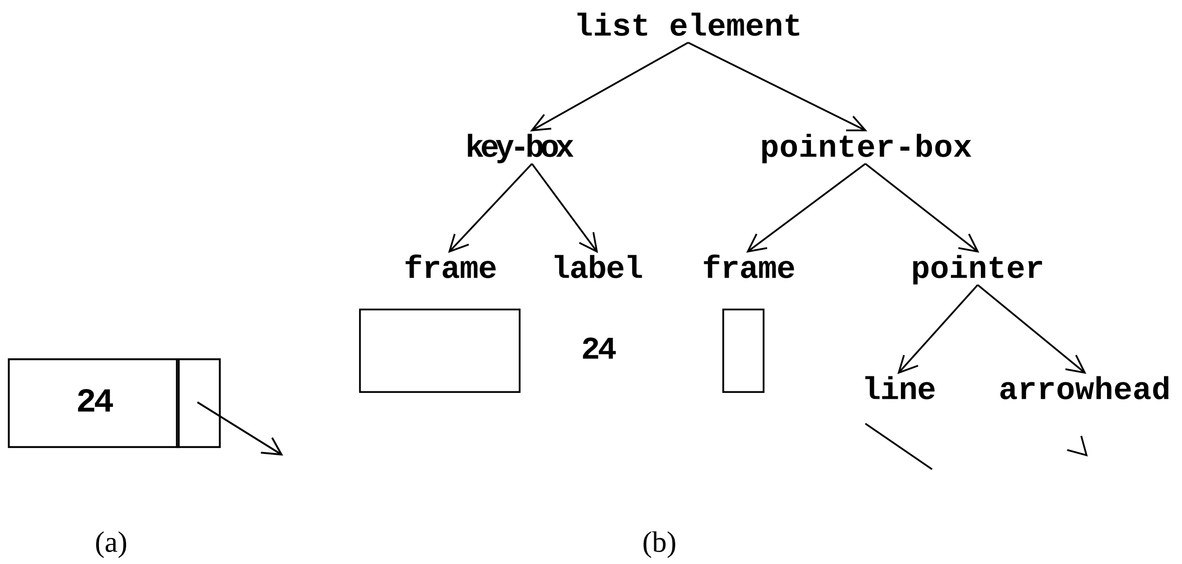


Figure 4: A list element (a) and the objects used to build this list element (b).

and then load it into an application or back into Lapidary.

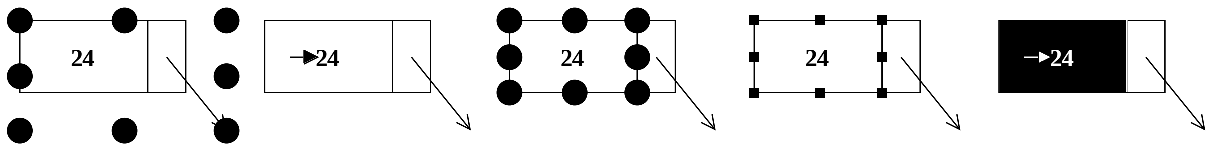
4.2 Object Composition

Lapidary provides a composition mechanism that allows objects to be collected into aggregate objects (also called ‘‘groups’’ or ‘‘collections’’). These aggregate objects can be used as prototypes, and instances can be made of these objects, just like any other object. For example the list element in Figure 3 has a top level aggregate containing a key-box and a pointer-box (Figure 4). The key-box is also an aggregate containing a frame and a label. Similarly, the pointer box is an aggregate containing a frame and a pointer. Finally the pointer is an aggregate consisting of a line and an arrowhead.

4.3 Novel Interaction Techniques

Unlike simple drawing editors such as MacDraw, Lapidary allows the objects within an aggregate to be selected and edited. For example, it allows the designer to constrain the size of children objects to the size of their aggregates. To allow the children of aggregates to be selected, Lapidary uses a number of novel interaction techniques. First, the user can choose either a ‘‘leaves’’ selection mode, which causes Lapidary to select leaf elements of an aggregate, or a ‘‘top-level objects’’ mode, which causes Lapidary to select top-level aggregates (objects that do not belong to an aggregate will be selected in either mode). To aid the user in determining whether they have selected a leaf or aggregate element, Lapidary uses different types of selection handles—rectangular handles for leaf elements (Figure 5.d) and circular handles for aggregates (Figures 5.a and 5.c). If the object is too small to accomodate 8 selection handles, either a thin or thick-lined arrow is used to highlight the selection (Figure 5.b), depending on whether the object is a leaf or aggregate object.

Once an object is selected, the user can cycle through the aggregate hierarchy by clicking



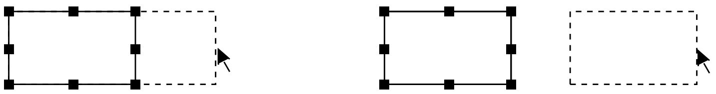
(a) (b) (c) (d) (e)

Figure 5: As the user repeatedly clicks the mouse button over an object, the selection cycles through the aggregate hierarchy shown in Figure 4. Initially the list element is selected (a), then the label (b), and finally the key-box (c). (a)-(d) represent various types of selection handles—circular handles for aggregates, (a) and (c), arrows for undersized objects (b), and rectangular handles for leaf objects. (e) illustrates the selection of a covered object. The label is covered by an xor feedback object, so the feedback object would be the initial selection. Pressing an appropriate mouse button allows the selection to be pushed down to the covered label.

repeatedly on the object. Each click moves the selection one level higher in the hierarchy, until the top-most level is reached, at which point the selection process cycles back to a leaf (if the selection is initially a top-level object, the next click cycles to the leaf). For example, when the user clicks on the label shown in Figure 14.a, and Lapidary is in "top-level objects" mode, the entire list element is selected (Figure 5.a). If the user clicks on the label again, the label is selected (Figure 5.b). Clicking once more with the mouse causes the key-box to become selected (Figure 5.c). Finally, one more click causes the list element to be selected, at which point the cycle repeats itself. In "leaves" mode, the label would be the first object selected, then the key box, and finally the list element.

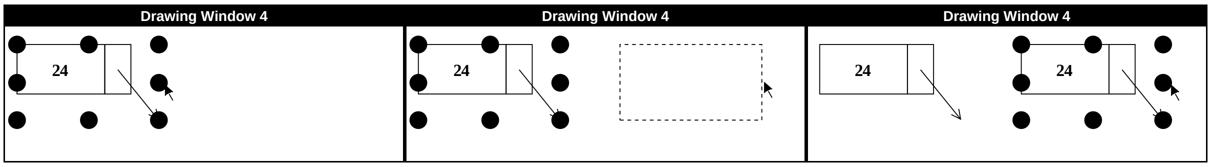
The user can also select covered objects without having to go through a series of "send to back" and "bring to front" operations (Figure 5.e). By pressing an appropriate mouse button, Lapidary allows a selection to be forced downward to a covered object. Once the covered object has been selected, the designer can cycle through its aggregate hierarchy, just like any other selected object. Finally, Lapidary supports a number of common selection operations, such as add to selection, remove from selection, and select all objects in a region swept out by the mouse. In each of these operations, leaves or top-level aggregates are chosen based on the current selection mode.

Once an object is selected, several other novel operations are available to the designer. Most drawing editors support constrained resize operations that change only one dimension of an object, either its width or height, if a selection-handle that is on a side is chosen (Figure 6.a). Lapidary supports this operation, as well as constrained move operations. Thus if the designer clicks on the selection handle on the right side of a rectangle, the rectangle will only move in the x direction (Figure 6.b). Lapidary also supports duplicating and instancing operations on the selection handles. If the user clicks the appropriate mouse button while over a selection handle (Figure 7.a), either a duplicate or instance of the object will be created and the user can move the object to the appropriate location in the window, or even to another window (Figures 7.b and 7.c). The selection handle constrains the initial movement of the new object, which facilitates the alignment of objects. Allowing the user to create new objects directly from the selection handles simplifies the duplication/instancing process. In other editors, the user must first pull down a menu or hit a keyboard accelerator, and then locate and move the new object with the mouse.



(a) (b)

Figure 6: Most drawing editors support constrained resize, in which only one dimension of an object is resized if one of the selection handles along the sides is chosen (a). Lapidary also supports constrained move, in which the object is moved along only one axis if one of the selection handles along the sides is chosen (b).



(a) (b) (c)

Figure 7: Objects can be duplicated or instanced by clicking on one of the selection handles (a), dragging the new object to the appropriate location (b), and dropping it (c).

Finally, to reduce ambiguity in operations that require distinct types of arguments, Lapidary provides two types of selections—primary and secondary selections. Primary selections are denoted using blackened selection handles, and secondary selections are denoted using transparent selection handles (Figure 9). The full range of selection operations—select, add to selection, deselect, remove from selection, select/deselect in region, select covered object, and cycling through the aggregate hierarchy—are provided for both types of selections (typically primary selections are made using the left mouse button and secondary selections are made using the middle mouse button). Two examples of operations that require primary and secondary selections are 1) the application of constraints, which need to distinguish between the object that a constraint should be applied to and the object that should be referenced in the constraint; and 2) the creation of parameters, which need to distinguish between the top-level object that contains the parameter and the child that actually uses the value of the parameter. Operations that do not require two distinct types of arguments ignore the distinction between primary and secondary selections, and apply equally to both types of selections. For example, a coloring operation will color all selected objects, regardless of whether they are a primary or secondary selection. Providing two distinct selection types reduces the ambiguity that can occur in other applications. For example, an inexperienced user of MacDraw generally cannot predict which object will be moved in order to align two selected objects. In contrast, Lapidary always moves the primary selection in order to accomplish an alignment. Distinct selection types also allow users to select arguments in any arbitrary order, which better adheres to the style of direct manipulation programming. For example, Lapidary allows a user to implement an alignment constraint by making a primary and a secondary selection in any order. In an application that does not provide multiple selection types, the user might have to follow a fixed order, such as first selecting the object to be aligned, and then selecting the reference object that it should be aligned to.

5 Constraints

Constraints are useful for specifying both the static presentation of an application and the dynamic behavior of the run-time application graphics. For example, the designer might specify that an arrow should be connected to the sides of boxes as shown in Figure 2. As described in Section 7, constraints can also be used to attach feedback objects to the mouse. Garnet stores mouse positions in slots called either `points` or `box`, depending on whether the object is a line or a box-like object. By attaching constraints that reference these slots to the position slots of a feedback object, the feedback object can be made to automatically track the mouse.

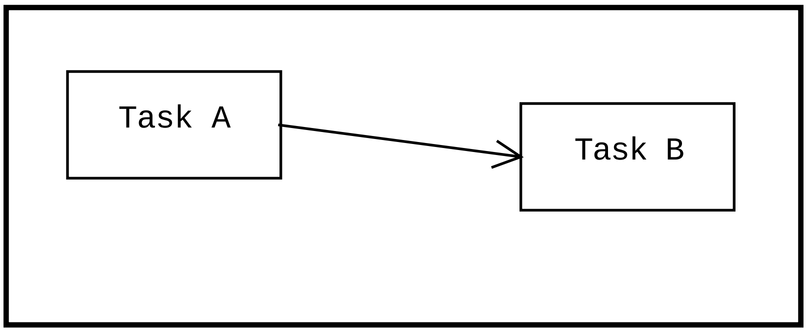


Figure 8: An arrow connected to two example boxes.

Constraints are also useful for aligning the parts of an object. In Figure 3, the designer has used constraints to 1) center the list element's key within a rectangle; 2) center one of the endpoints of the list element's ''next'' pointer within a second rectangle; and 3) align the pointer's rectangle next to the key's rectangle and make the height of the pointer rectangle be the same as the height of the key rectangle. If the list element is moved, all of its parts will automatically move with it and stay appropriately aligned. Finally, constraints ensure that the static parts of an interface look attractive. For example, it is trivial to ensure that objects, such as the object and delete/quit menus in Figure 2, are lined up correctly, or are exactly centered. This is in contrast with interfaces created by some other UIMSs, where objects must be aligned by hand with the mouse and may look sloppy. It is also often easier to use constraints than gridding to align objects, since the desired locations will not always end up on a fixed grid.

If the desired constraint is one of a standard set, then it can be specified easily using the Lapidary iconic constraint menus (Figures 9 and 11). There are two such menus—one for specifying constraints between box-like objects and one for specifying constraints between line-like objects and other objects (either line-like or box-like). The box-constraint menu supports having objects be connected on their edges or in the middle, with optional offsets. The sizes of objects can also be related. The line-constraint menu supports having the endpoints of lines connected to the corners, edges, or center of box-like objects, and to the endpoints of other lines. It also supports connecting the corners, edges, or center of box-like objects to the endpoints of lines. Like the box-constraint menu, the line-constraint menu supports optional offsets for the connection points. Experience with Peridot [13] demonstrated that these simple types of constraints make up the vast majority of those needed in typical user interfaces. When the designer needs a constraint that cannot be expressed using one of the standard constraints, the designer can bring up Garnet's C32 spreadsheet editor (Figure 12) [19].

To specify a constraint, the designer selects one object as the primary selection, another object as the secondary selection, and then selects the constraint to apply (see Figure 9). The primary object is then made to depend on the secondary object (the primary object changes, and the secondary object does not change).

5.1 Box Constraints

The `left` (horizontal row of buttons at the top), `top` (vertical row of buttons at the left), `width`, and `height` each have their own section of the box constraint menu. For the `left` and `top` slots, the buttons represent the primary selection and the large box represents the secondary selection. The buttons for the `left` slot constrain the primary selection to be at the left-outside, left-inside, center, right-inside, and right-outside of the secondary selection (Figure 10). The `unconstrain` button destroys the constraint, if one exists, in the `left` slot of the primary selection. The offset can be used to adjust how close the objects are (the default is zero). If the centered button is selected, the `offset` button changes to one labeled `percent` and selects what percent of the way across the secondary selection the primary selection should be. The default is 50%, which is directly in the center. In Figure 9, the primary selection is the white rectangle, and it is to the right of the gray rectangle, offset by 20 pixels.

The constraints for the `top` slot have a similar interpretation as the constraints for the `left` slot. For the `width` and `height`, the constraints are either on or off, and these can be modified either by an offset (`A.Width = B.Width + offset`), a percentage (`A.Width = percent * B.Width`) or both. In Figure 9, the white rectangle has no constraints on its width, and its height is 33% of the height of the gray rectangle.

Finally, the text boxes next to the `:left`, `:top`, `:width`, and `:height` labels display the current values for a selected object's position and size slots. They also allow the user to enter specific values for an object's `left`, `top`, `width`, and `height`. This option is useful when the designer wants specific values for these slots, rather than the approximate values that are obtained using direct manipulation or the computed values that are obtained using constraints.

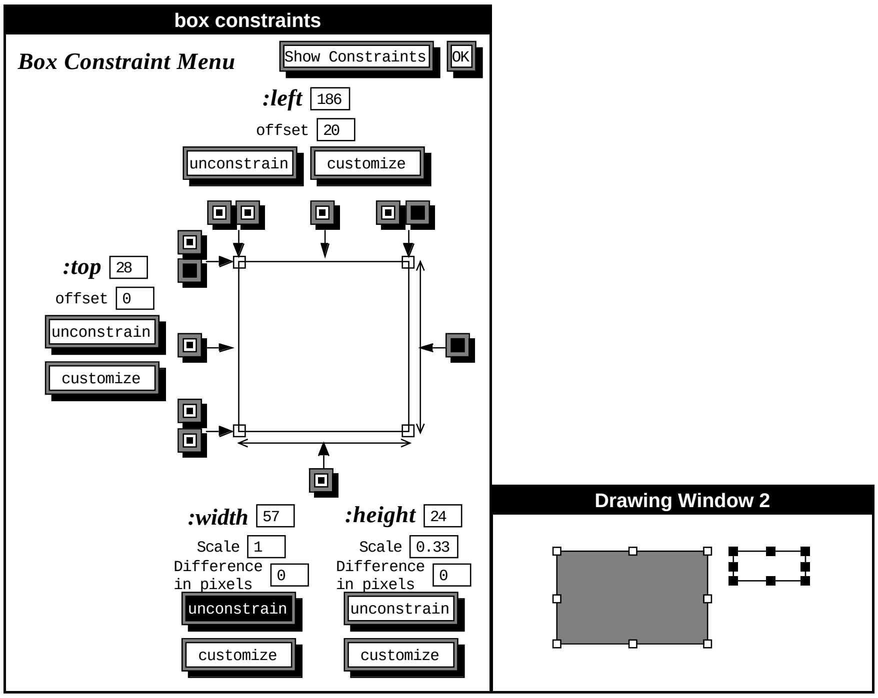
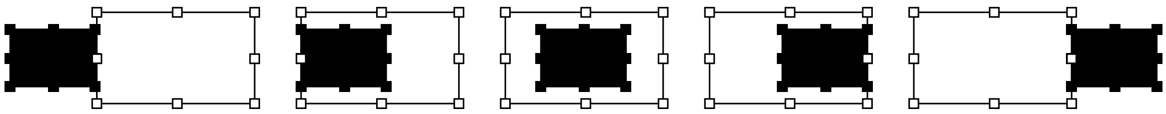


Figure 9: The Lapidary constraint menu for box-like objects on the left, and the drawing window on the right. The white rectangle in the drawing window is the primary selection and the gray rectangle is the secondary selection.'' In the section labeled ''left'' of the constraint menu, the darkened box shows that the white rectangle is constrained to be offset from the right of the gray rectangle by 20 pixels. Similarly, the ''top'' section of the constraint menu shows that the white rectangle is aligned at the top-inside of the gray rectangle, and the height section shows that the white rectangle is 33% as tall as the gray rectangle. The white rectangle's width is not constrained. If the gray rectangle changes, the white rectangle will be adjusted automatically.



(a) (b) (c) (d) (e)

Figure 10: The buttons for the `left` slot in the box constraint menu allow objects to be constrained to the (a) left-outside, (b) left-inside, (c) center, (d) right-inside, and (e) right-outside of an object.

5.2 Line Constraints

The line constraint menu supports three types of constraints: line to box, line to line, and box to line. To make it clear what type of constraint is being applied, the line constraint menu displays the type of the primary and secondary selections (either a line or a box). It has two sections: one for the primary selection and one for the secondary selection. Each section displays both a line and a box. The line objects have buttons indicating that endpoints can be constrained and the box objects have buttons indicating that edges, corners, and the center can be constrained. When an object is selected, the line constraint menu grays out whichever feedback object does not correspond to the type of the selected object. In addition, if the selected object is a line, then the appropriate feedback line is rotated, so that its alignment matches the alignment of the selected line. This rotation makes it clear which endpoint of the selected line is being constrained, or being used as a reference in a constraint. To reinforce the notion that the feedback objects in the primary selection section denote primary selections, the boxes in the middle of the buttons are colored black. Similarly the boxes in the middle of the buttons in the secondary selection section are white.

To create a constraint, the designer pushes the appropriate button on the enabled feedback objects in both the primary selection and secondary selection areas. For example, in Figure 11, the designer has attached the rightmost endpoint of an arrow to the left corner of a circle. The connection between two points can be fine-tuned using the `x-offset` and `y-offset` fields. For example, if the designer wanted a line to be connected to the left side of a box, 10 pixels from the top, the designer could constrain the line to the left corner of the box, and enter a y-offset of 10. The `x1`, `y1`, `x2`, and `y2` boxes show the current values of a line’s endpooints and allow the designer to specify an exact pixel position for the endpoints of a line. The `unconstrain` button destroys the constraints on the endpoint of a line or the position of a box.

5.3 Custom Constraints

If the designer wants to express a relationship that cannot be specified using the standard choices provided in the box and line constraint menus, the designer can select the `Customize` buttons in the constraint menus. Lapidary will bring up Garnet’s C32 spreadsheet editor [19] and a formula window for the desired slot (Figure 12). The designer is allowed to type in an arbitrary CommonLisp expression specifying the formula. This expression can use conditionals, loops, local variables, and any other Lisp form. Objects can be selected using the mouse from either the drawing window or the spreadsheet during this process, and the system will automatically include references to them in the constraint.

As an example of where a custom constraint is needed, Figure 13.a shows an interface where the ‘‘from’’ endpoints of the arrows were attached to the right sides of boxes and the ‘‘to’’ endpoints were attached to the left sides of boxes using the standard line constraint menu. However, when boxes are moved to different parts of the display, as in Figure 13.b, the placement of the arrows looks awkward. In Figure 13.c, the constraints on the endpoints of the line have been changed so that they appear at the most appropriate corner or edge of a box. The choice of where to attach an arrow is made based on whether the ‘‘from’’ object is above, below, to the left of, to the right of, or overlapping an object. The Lisp code for the custom formula that computes the y-coordinate of the second endpoint is shown in Figure 12.

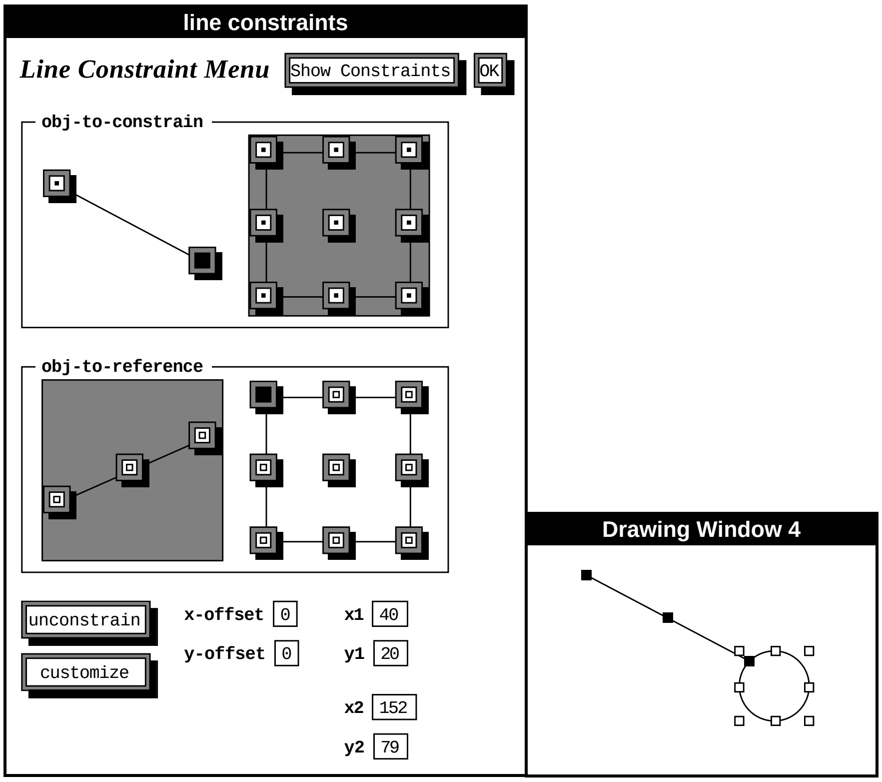


Figure 11: The Lapidary constraint menu for line-like objects on the left, and the drawing window on the right. The line in the drawing window is the primary selection and the circle is the secondary selection.'' In the primary-selection section of the constraint menu, the line feedback object has been rotated so that it has the same orientation as the selected line, and the box feedback object has been disabled (grayed-out). In the secondary-selection section the line feedback object has been disabled since the secondary-selection is a box-like object. The darkened buttons on the right endpoint of the line feedback object and the left corner of the box feedback object indicate that the right endpoint of the line is attached to the upper left corner of the circle.

Clearly, writing Lisp code is not a job for user interface designers that are not programmers, but we did not feel that there was any reason to try to invent a special language that would be more accessible. The most common relationships are trivial to specify using the iconic dialog boxes, and C32 provides limited syntax-directed editing for the more common Lisp arithmetic commands. More complex constraints are not likely to be attempted by non-programmers anyway. Since the programmers are likely to already know Lisp (because the application is written in it), there is a tremendous disadvantage to making them learn a new language [26].

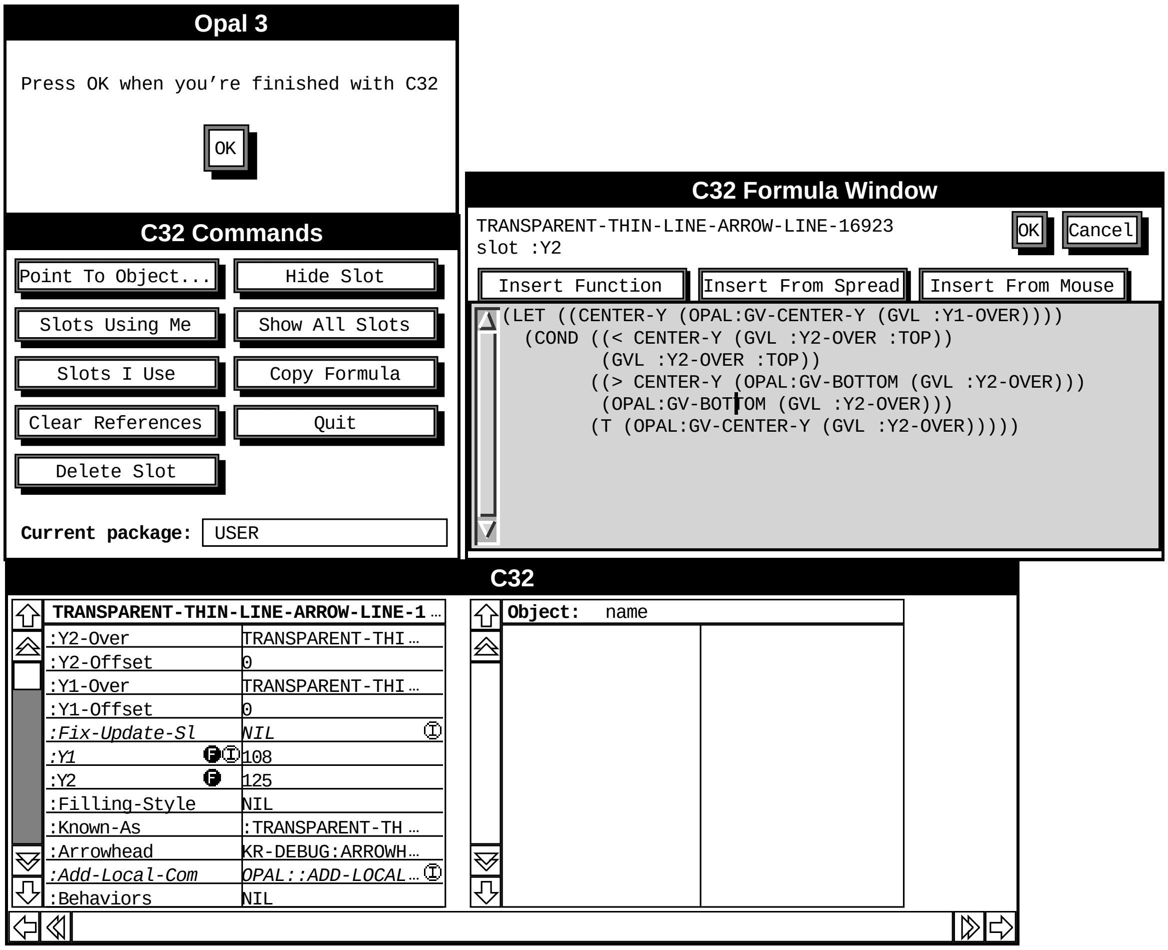


Figure 12: The various C32 windows. The main spreadsheet window (c) displays the slot values of selected objects. The user can edit these values and the changes will be reflected immediately in Lapidary. The `I` icons denote inherited values and the `F` icons denote values computed by formulas. To examine or edit a formula, the user clicks on the appropriate `F` icon and C32 brings up the formula window (d). Here the user is editing the formula in the `:y2` slot of an arrow, so that the arrow will connect to the appropriate edge or corner of a box. The formula window allows the user to enter arbitrary Lisp code. When the user hits `OK`, the formula is installed immediately on the appropriate slot and its effects are visible in the Lapidary drawing editor. The text buttons window (b) lists C32's commands.

5.4 Constraints on Non-Numeric Properties

Constraints can be used to compute the value of any property of an object, not just the numeric ones. For example, the string for a label might be constrained to be the name of an object, or the color of a rectangle might be computed based on the value of a status variable. For properties that have dialog boxes, such as line-style, text, and filling-style, a formula button can be pressed and Lapidary will display both C32 and a formula window for the appropriate slot. In the case of properties that do not have dialog boxes, such as the `visible` slot, the user can pull down the ''constraints'' menu and bring up C32 explicitly. It is also easy to add new application-specific slots to objects and to define constraints for

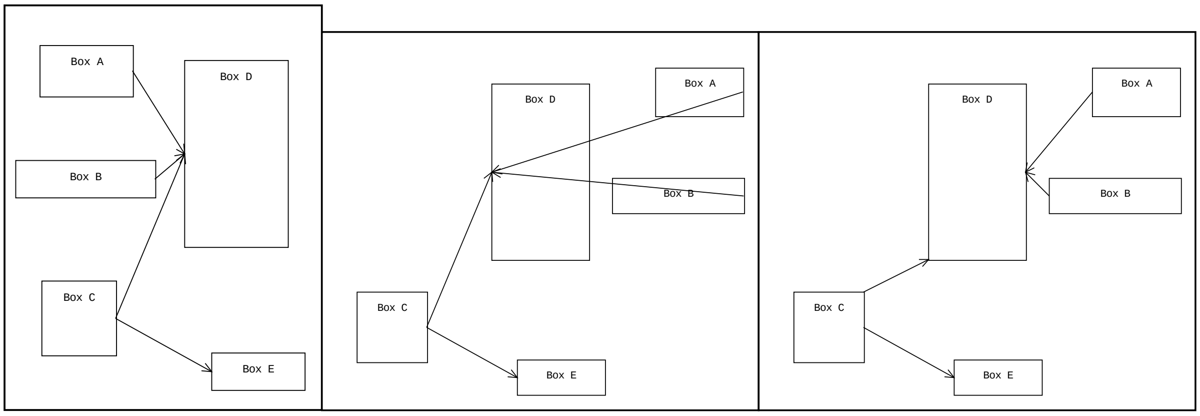


Figure 13: Arrows are connected to the centers of the boxes by constraints so they stay attached even when the objects are moved. In (a) and (b), the endpoints of the arrows are constrained so that the ‘‘from’’ endpoints are connected to the right side of a box and the ‘‘to’’ endpoints are connected to the left side of a box. When boxes are moved, these constraints can lead to an awkward positioning of the arrows, as in (b). In (c), custom constraints have been placed on the endpoints of the arrows so that they can be attached to the appropriate corner or edge of a box, depending on the relative positions of the ‘‘from’’ and ‘‘to’’ boxes.

these slots. Each of these constraints can be added by bringing up the C32 spreadsheet editor, entering the slot name, and then bringing up a formula window.

6 Generalizing the Constraints

In order for the graphical objects to be useful at run-time, the specific constraints must be generalized to work on run-time objects, rather than on the specific example objects used in the editor. For example, in Figure 11, the line is attached to an example circle, and the constraints refer to this specific circle object. When instances of this line are created, they will inherit the constraints, just like ordinary values. However, since these constraints reference the example circle, the instances will be connected to the example circle rather than arbitrary circles created at run-time. Thus the constraints must be generalized so they can reference arbitrary circles.

Garnet makes this easy by supporting pointer variables in constraints [30]. Pointer variables allow constraints to indirectly reference other objects rather than referencing them through hard-coded direct object references. For example, instead of representing the constraints on the arrow’s first endpoint as[3]

```
arrow:
    x1: circle_1011.center-x + (circle_1011.radius * cos (.75 * π))
    y1: circle_1011.center-y + (circle_1011.radius * sin (.75 * π))
```

[3]For the sake of readability, we are expressing constraints using a conventional notation instead of a Lisp notation. In Garnet, the constraint `circle_1011->right` would be written `(gv circle_1011 :right)` and the constraint `self->x1_ptr->right` would be written `(gvl :x1_ptr :right)` where `gv` stands for *g*et *v*alue and `gvl` stands for *g*et *v*alue through *l*ink. In Garnet, colons are placed before slot names but are dropped in this paper.

Lapidary represents them as

```
arrow:
    x1: self->x1_ptr->center-x + (self->x1_ptr->radius * cos (.75 * π))
    y1: self->y1_ptr->center-y + (self->y1_ptr->radius * sin (.75 * π))
    x1_ptr: circle_1011
    y1_ptr: circle_1011
```

`self` is a special reference to the object containing the constraint. Thus `self->x1_ptr->center-x` would find the value of the `x1_ptr` slot in `arrow`, and then retrieve the value of `center-x` from the object in the `x1_ptr` slot.

Pointer variables allow Lapidary to easily generalize constraints by storing the references to example objects in pointer variables. When instances of the constraints are created, the instances can be made to reference arbitrary objects by making the pointer variables point to the desired objects rather than the example objects. For example, when the example arrow is cloned at run-time, its clones will inherit instances of the constraints on `x1` and `y1`. The application simply sets the `x1_ptr` and `y1_ptr` slots to an appropriate circle object, and the constraints ensure that the cloned arrow will be attached to the upper-left corner of this circle object.

Thus pointer variables automate part of the process of converting the example object to a prototype. It is important to emphasize that Lapidary makes these transformations automatically. The user interface designer never sees any of this code. Even if the designer creates custom constraints using C32, Lapidary ensures that pointer variables are used by converting direct references to indirect references. For example, when the designer points at example objects with the mouse, Lapidary creates a pointer variable for the reference and inserts a reference to the example object in the newly-created pointer variable.

It is also important to emphasize that the designer has an opportunity to change the Lapidary-generated names for pointer variables and to merge pointer variables that reference the same example object into one pointer variable. This renaming is important so that the application knows which pointer variables to set and so that it does not have to set lots of identical pointer variables. For example, Lapidary gives the designer the opportunity to merge the `x1_ptr` and `y1_ptr` pointer variables, and to assign the merged pointer variable a new name, such as `from_obj`. This renaming process is described in Section 6.2.2.

6.1 Integrating Constraints with Object Hierarchies

Another problem that Lapidary automatically handles is the integration of constraints with object hierarchies. When an instance of an aggregate object is made, the pointer variables of constraints in the children must be adjusted so that they point to objects in the instance rather than objects in the prototype. For example, suppose that a labeled box consists of a rectangle and a text string, and that the text string is centered in the rectangle. When an instance of the labeled box is created, the constraints in the instance’s text string should refer to the instance’s rectangle, not the prototype’s rectangle. Pointer variables partially handle this problem, because references are not hard-coded, but they do not completely handle the problem, because the correct references need to be placed in the pointer variables.

The rest of this problem is solved by automatically creating pointer variables for each child that points back to the parent, and pointer variables for the parent that point to each child. The pointer variables back to the parent are called `parent`, while the names of the pointer variables to the children are derived from the names of the children. For example, the two components of key-box in Figure 14 have `parent` pointers to key-box, and key-box has two pointers, named `frame` and `label`, that point to its two children.

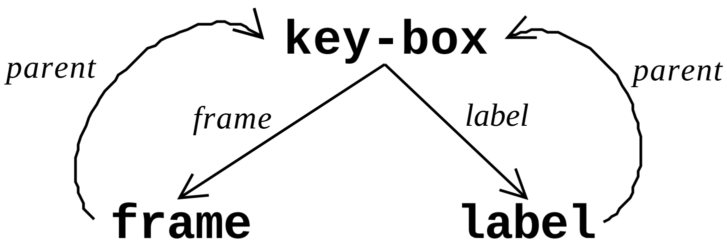


Figure 14: Pointers that Garnet creates for the key-box object in Figure 14. Both children have back pointers, labeled `parent`, to their parent, and `key-box` has pointers to its children labeled `frame` and `label`.

These pointers allow Lapidary to create paths of pointers that navigate through the aggregate hierarchy. For example, the string can use the path `self->parent->frame` to reference the appropriate rectangle. Thus, when an object belongs to the same aggregate hierarchy as the object it references in a constraint, Lapidary creates a path to the referenced object and stores the path, rather than a direct reference to the object, in a pointer variable. When the referenced object is not in the same aggregate hierarchy, a direct reference is placed in the pointer variable, as described earlier. The constraints for the text string in the key box would therefore be written as:

```
center = self->center_ptr->center
center_ptr = self->parent->frame
```

Thus when an instance of an aggregate object is created, the Lapidary-generated paths ensure that the constraints in the children refer to the appropriate objects.

It may seem inefficient to have two levels of indirection in a constraint: first a reference to a pointer variable, which in turn follows a pointer path. For example, the constraint for `center` follows `center_ptr`, which then follows the path `parent->frame`. It would be more efficient if, at run-time, the path were directly encoded in the constraint as `self->parent->frame->center`. Lapidary recognizes that this improvement can be made, so when it saves an object to a file, it directly encodes the path in constraints and gets rid of the corresponding pointer variable.

6.2 Parameter Inferencing

The generalization of an example object into a prototype object is not complete until Lapidary has identified which slots in the example should be parameters that are set at run-time by the application. There are two possible types of parameters: attribute parameters and pointer-variable parameters. Attribute parameters control the appearance of an object, and include attributes such as line-style, filling-style, and text string. Pointer-variable parameters are used in constraints and determine the objects this object will be constrained to in the application. Lapidary currently requires the designer to explicitly designate attribute parameters and attempts to infer pointer-variable parameters.

All parameters, regardless of whether they are an attribute or a pointer-variable parameter, should be placed in the top-level aggregate. This design allows the application to deal only with the top-level aggregate and obviates the need for the application to know about the aggregate's structure. Thus the object can be modified without changing the application and the application programmer does not have to write code that traverses the aggregate hierarchy. For example, in Figure 4, one of the parameters is the value of the label. If the label's value is a top-level parameter (e.g., a parameter to `list element`), then `list-element` can be arbitrarily changed without affecting the application. However, if the application had to directly set the value of label, then the application programmer would have to determine the structure of `list-element` and write the appropriate code to traverse the aggregate hierarchy to find the label. If `list-element` was later changed, the traversal code might be made obsolete and have to be changed as well.

If the parameters are specified at the top-level, there must be some mechanism for communicating their values to the appropriate objects. Lapidary uses constraints for this purpose. For each attribute or pointer-variable that should be made into a parameter, Lapidary creates a constraint that retrieves the attribute's or pointer variable's value from a top-level parameter variable that is defined by the designer. The way in which Lapidary creates parameters is described in the next two sections.

6.2.1 Attribute Parameters

To create an attribute parameter, the designer makes the object whose attributes are being made into parameters the primary selection and makes the top-level object that the attributes should retrieve their values from the secondary selection. For example, if the user wants to parameterize the label and font of the label object shown in Figure 4, the user would make the label object be the primary selection and the list element be the secondary selection. Once the appropriate objects have been selected, the designer chooses a ‘‘parameters’’ option, which brings up the dialog box shown in Figure 15. The dialog box lists the attributes that can be made into parameters and provides type-in text boxes that the designer can use to indicate which slots in the secondary selection the attributes should retrieve their values from. In Figure 15, the label's `string` slot retrieves its value from list element's `value` slot, and the `font` slot retrieves its value from the list element's `font` slot.

Once the designer identifies an attribute as a parameter, Lapidary inserts a constraint into the attribute's slot that retrieves the slot's value from the appropriate slot in the secondary selection. This constraint uses a path to traverse the aggregate hierarchy and reach the secondary selection. For example, since the label in Figure 4 is a descendent of list-element, Lapidary would store the constraint `string =`

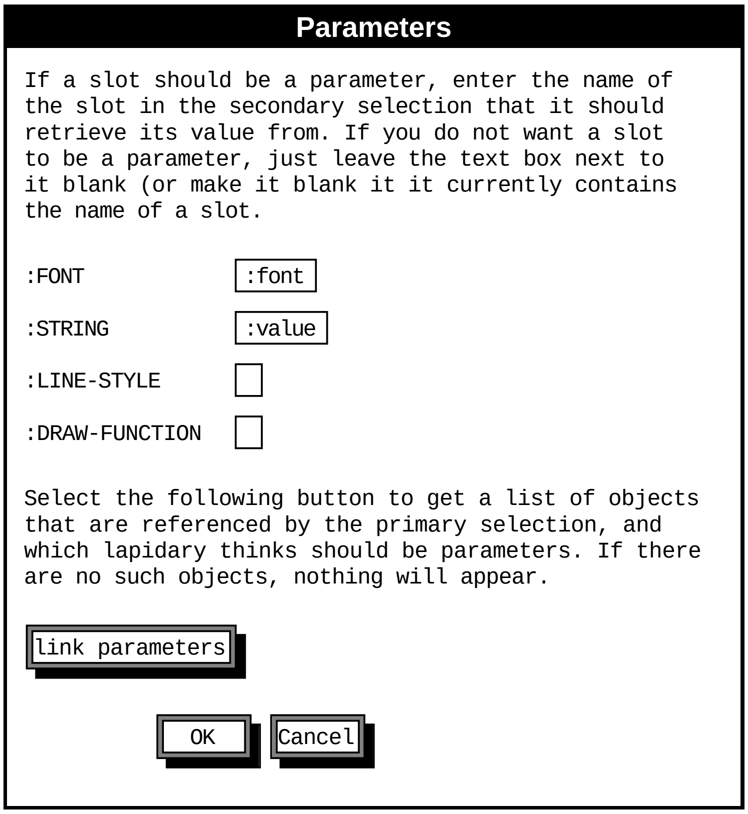


Figure 15: Attribute parameters dialog box

`self->parent->parent->value` in label's `string` slot. When an instance of list-element is created at run-time, the application can set the value of the label by placing an appropriate string in the `value` slot of the list-element instance.

Lapidary maintains a list of slots for each object that can be turned into parameters. If the user wants to parameterize a slot that is not displayed in the parameters dialog box, the user can bring up C32 and place a formula in the desired slot that retrieves its value from the top-level aggregate. C32 can also be used to create a formula that filters a value, for example, turning a boolean into a color.

6.2.2 Pointer Variable Parameters

Pointer variable parameters are created when the designer presses the link parameters button in the parameters dialog box or saves an object. As described previously in this section, Lapidary uses indirect references in its constraints so that it can easily generalize them. There are two types of indirect references—references to an object in the same aggregate hierarchy as the object containing the constraint and references to objects not in the same aggregate hierarchy as the object containing the constraint. In the former case, the pointer variable containing the reference holds a path to the referenced object. When instances of the prototype aggregate that contains the objects are created at run-time, the pointer variable will reference an appropriate instance object. Thus Lapidary assumes that these pointer variables should not be parameters.

In the latter case, the pointer variables that contain the reference point directly to the referenced object. Therefore, all instances of the prototype aggregate created at run-time will point to the same object. This may not be what the designer intended, and thus Lapidary assumes that these pointers may be parameters. Lapidary displays these pointer variables in the link parameters dialog box (Figure 16). For each such pointer variable, Lapidary displays the value of the pointer variable, the slots that reference the pointer variable, and a parameter name, if any, that the user has assigned to this pointer variable. The user can change this parameter name by editing it, or can indicate that this pointer variable should not be a parameter by making the parameter name blank. Any pointer variable that is converted to a parameter will be assigned a null value when it is written out, since the example object it points to in Lapidary cannot be assumed to exist at run-time.

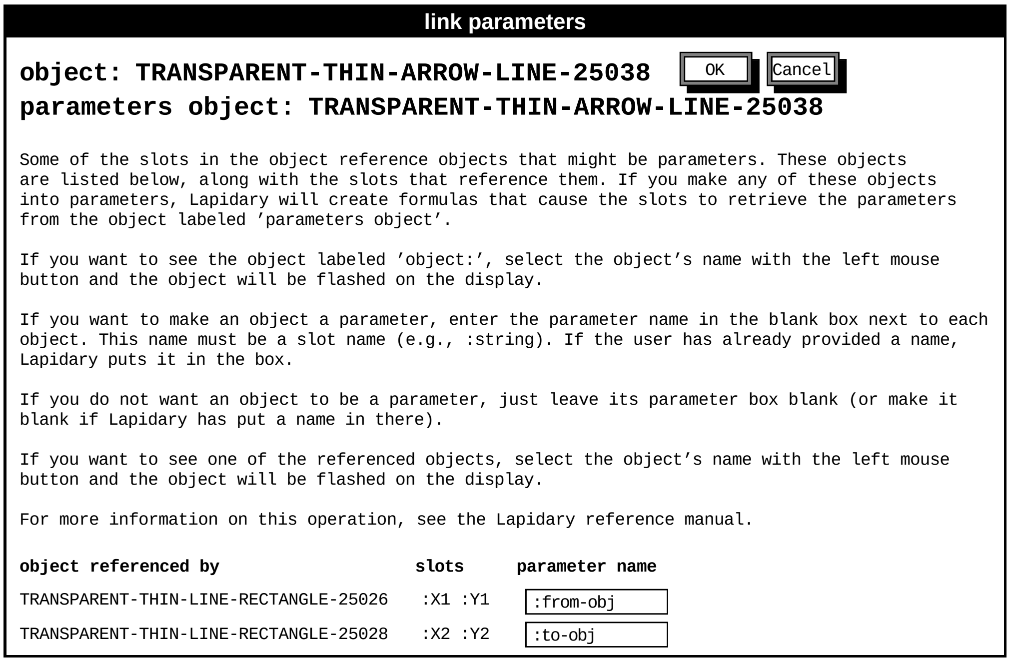


Figure 16: Link parameters dialog box

For example, the link parameters box in Figure 16 reflects the inferences Lapidary makes when a designer saves the arrow that is connected to the two boxes in Figure 8. The designer used constraints to attach the ends of the arrow to the sides of the two boxes, but these two boxes are not part of the arrow's prototype. Lapidary notices that the slots `x1`, `y1`, `x2`, and `y2` references these boxes and infers that these boxes should be parameters. The designer has confirmed these inferences by typing in parameter names for the boxes, `from_obj` and `to_obj`. When an application creates an instance of this arrow, it can instantiate the `from_obj` and `to_obj` slots with any arbitrary objects, and the instance arrow will attach itself to the objects in the appropriate way.

For each pointer variable parameter, Lapidary creates a slot with the name of that parameter, and constructs a constraint that retrieves the value of the slot from the top-level aggregate (the constraint has the effect of promoting the parameter to the top-level aggregate). The constraint accesses the top-level aggregate by using parent pointers to climb the aggregate hierarchy. Lapidary also substitutes the new pointer variable name in the constraints for the previous pointer variable names that referenced the example objects. For example, the constraints on the arrow's left endpoint were originally represented as:

```
arrow:
    x1: self->x1_ptr->right
    y1: self->y1_ptr->center-y
    x1_ptr: box1
    y1_ptr: box1
```

Once the parameter pointer slots have been created and substituted into the constraints, the constraints are represented as:

```
arrow:
    x1: self->from_obj->right
    y1: self->from_obj->center-y
    from_obj: null
```

Notice that the two slots, `x1_ptr` and `y1_ptr`, that pointed to the same object have been merged into one pointer variable called `from_obj`. In this case the arrow was the top-level object, so no constraint was created for the `from_obj` slot. If the arrow had been the child of another object, a constraint of the form `self->parent->from_obj` would have been placed in the arrow's `from_obj` slot, and a `from_obj` slot in the arrow's parent would have been created and set to null.

7 Creating Interactive Behavior

7.1 Interactors Model

Constraints provide one mechanism for defining the dynamic behavior of objects. However, they do not specify how these objects should respond to input events from various devices, such as mice and keyboards. Garnet provides this capability by encapsulating the most commonly occurring interactive behaviors into a small number of ''interactor'' objects [16, 18]. The interactors supported by Lapidary include:

- Choice-of-Items - for selecting one or more items from a set of items using the mouse (e.g., for menus and buttons, or for selecting an object in a graphics editor),
- Move-Grow-Interactor - for changing the size and/or position of objects with the mouse,
- New-Point-Interactor - for entering new points with the mouse (e.g., for creating new objects),
- Angle-Interactor - for measuring angles as the mouse moves around a point (e.g., for circular gauges),
- Edited-Text-String-Interactor - for entering edited single or multiple lines of text using the keyboard.

The interactors represent pure input device behaviors, and are devoid of any graphical presentation.[4] A rich variety of parameters allow the designer to customize the behaviors in many different ways. Parameters may be divided into two types: behavior-defining parameters which specify properties of the behavior, and graphics-defining parameters which link graphics to the behavior.

[4] Therefore, this use of the term ''interactor'' is different from Cardelli's [3]. In his system, an ''interactor'' is what is here called a widget—a combination of graphics and behavior.

The behavior-defining parameters tend to differ with each interactor. For the choice-of-items interactor, the behavior-defining parameters allow the designer to specify which mouse button starts and stops the action; whether single or multiple objects can be selected; whether the item under the mouse is added, removed or toggled in the set of objects selected; and an application procedure to be called when the interactor is complete (other parameters can be specified as well—for a complete discussion, see [18, 21]). It is useful to note that an attached procedure is only needed if a semantic action is required. Usually, all graphical updates are handled automatically by the constraints.

Each interactor also has a number of graphical parameters that need to be specified. These parameters include a `start-where` parameter which lists the objects that the interactor should operate on, and an `interim-feedback` parameter that allows the designer to specify how graphics should change as the interactor operates, so the user can see what the interactor is doing. In addition, the choice-of-items interactor has a `final-feedback` parameter that allows the designer to specify how graphics should change to indicate the choice the user makes, and the text, move-grow, and angle interactors have an `obj-to-change` parameter that indicates which object should be modified by the interactor. For example, in Figure 17.a, the radio buttons are the objects that a choice-of-items interactor would operate on (the `start-where`), a thickened circle denotes the item the mouse is currently over (the `interim-feedback`) and the black dot denotes the final selection (the `final feedback`). In Figure 18.a, the selection handles are the objects that a move-grow interactor would operate on, the dashed-line rectangle is the interim feedback, and the rectangle highlighted by the selection handles is the `obj-to-change`.

The interactors must communicate the information they compute to their graphical parameters so that the appropriate graphical response is provided. They do so by setting pre-specified slots in the graphical parameters. For example, the choice-of-items interactor makes the `obj_over` slots in the interim- and final-feedback objects point to the object currently selected by the interactor. The move-grow and new-points interactors set the `box` slot in a feedback object with a list that contains the computed position and size information for an object. When the move-grow interactor finishes, it also sets the `box` slot in the object being moved or resized with the computed position and size information. Finally, the angle interactor sets an `angle` slot and the edited-text-string interactor sets a `string` slot in the feedback object, and, when the interactor finishes, in the object that is being changed.

The graphical objects use this information to provide the appropriate pictorial feedback. The feedback objects for the choice-of-items interactor can use the `obj_over` slot to access the selected object's position and size in order to highlight the selected object. Similarly, the feedback for the move-grow and new-point interactors can use the `box` slot to set themselves to the appropriate position and size. Finally, the objects being changed can access the appropriate slots to set attributes that are being computed by the interactor, such as position, size, angle, and string. All of this accessing is accomplished by placing constraints in the appropriate attribute slots. These constraints access the slots set by the interactors, and if necessary, provide a filter for the information computed by the interactors. For example, if an object should not move outside of a certain region, the constraint could check whether the computed position was outside of the region, and, if it was, return a position inside the boundary of the region instead.

The separation of the behavioral specification from the graphical specification of the interactors makes them look-and-feel independent. It is possible to edit the behavior without changing the graphics, and it is possible to edit the graphics without changing the behavior. For example, the selection boxes in Figure 18 could be made hollow and larger, or even made into an aggregate of items without affecting the underlying move-grow behavior (Figure 18.b).

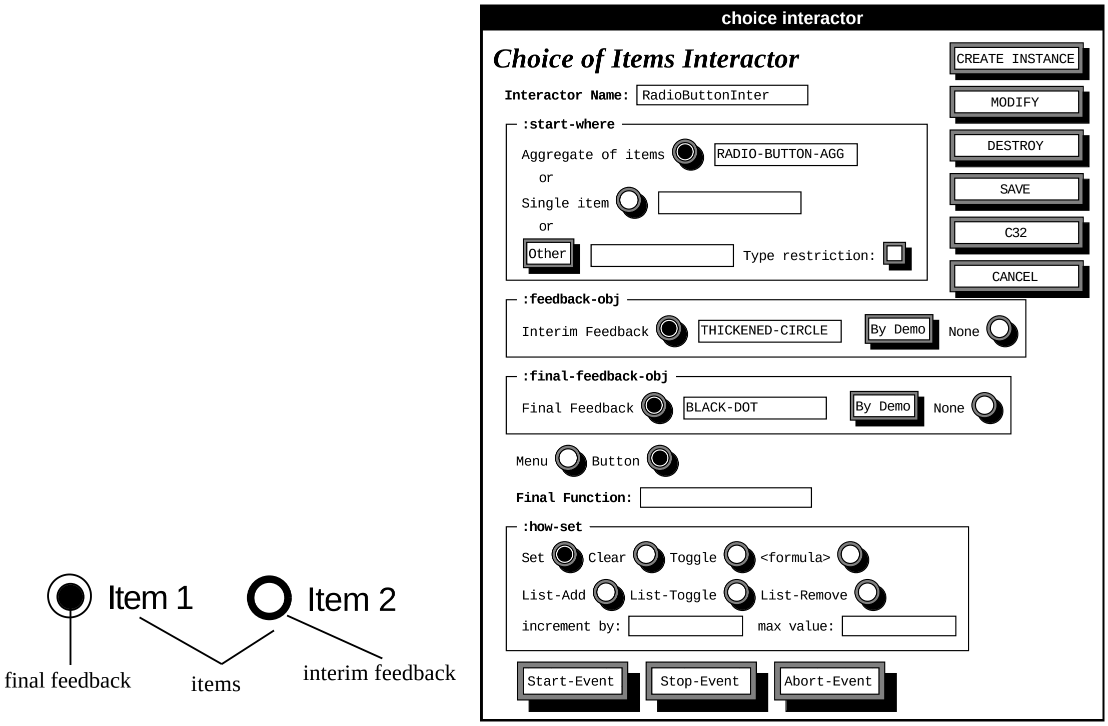
The interactors model also provides a rigorous, formal basis for specifying behaviors in Lapidary. Since there are only fives types of interactors and since the parameters are well-defined, Lapidary can easily provide dialog boxes for the user to fill in the desired values, or can attempt to infer the behavior and its parameters from the user's actions. Several of the interactors, such as the move-grow and new-point interactors, are specifically intended for specifying application-specific behaviors, while the choice-of-items, edited-text-string, and angle interactors are useful for specifying both widget and application-specific behaviors. This model contrasts favorably with the model used in many systems in which a set of pre-defined interaction techniques is tightly bundled with the graphics, making it difficult, if not impossible, to attach behaviors to *application* objects.

7.2 Dialog Boxes

Lapidary displays dialog boxes for each of the interactors that allows designers to enter both behavior and graphics parameters (see Figures 17 and 18). Since the interactors provide a large number of parameters, only the most frequently used ones are displayed in the dialog boxes. Ones that are not displayed can be set by bringing up the C32 spreadsheet editor and editing the appropriate slot. Most fields of the dialog boxes contain appropriate default values, either proposed by Lapidary or because the interactor was copied from a prototype.

The most interesting parameters from the standpoint of Lapidary are the graphical parameters. The designer can attach graphics to an interactor by selecting graphical objects in an editor window and then clicking on the radio button in the dialog box next to the appropriate graphical parameter (e.g., the start-where parameter that controls which objects the interactor operates on, the interim and final feedback parameters which control what type of feedback the user sees as the behavior is executing, or the obj-to-change parameter that controls which object the interactor changes). For example, to create the move-grow behavior shown in Figure 18, the user can pull up the move/grow dialog box, select the aggregate containing the selection handles and make it the start-where, select the dashed-line rectangle and make it be the interim feedback, and select the rectangle highlighted by the selection handles and make it the object to change.

The graphics that are associated with feedback objects and objects-to-change are especially interesting since they are generally example objects, and must be generalized in some fashion. Feedback objects must be generalized so that they can highlight any object operated on by the interactor. For the move-grow interactor, new-point, and edited-text-string interactors, the designer can simply draw the desired feedback object, select it, and indicate that it should be the interim-feedback object. These



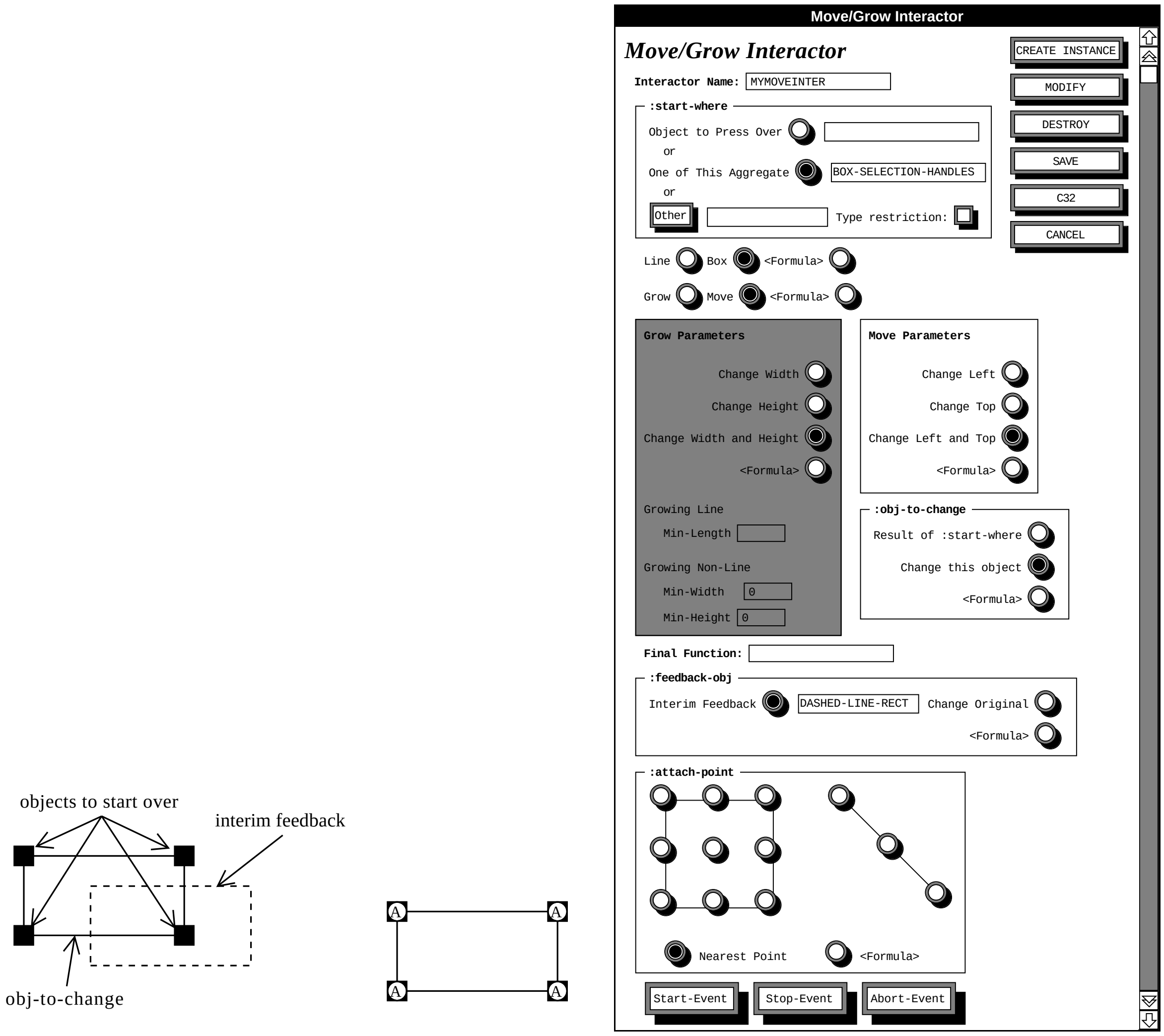
(a) (b)

Figure 17: (a) A Macintosh-like radio button, with the parts labeled as to the roles they play in the Choice-of-Items interactor; and (b) the dialog box for specifying the choice-of-items interactor.

interactors will set the `box` slot in the feedback object with a list that provides a computed size and position, so Lapidary places constraints in the position and size slots of the feedback object which reference the `box` slot. For example, it might place constraints in the `left` and `top` slots of the feedback object of the form:

```
left = first(self->box)
top = second(self->box)
```

For the choice-of-items interactor, the designer demonstrates how the feedback object's position and size should depend on the selected object by using constraints to position or size the feedback object with respect to an example item from the set of selectable items. The designer must do this for both interim and final feedback objects. For example, in Figure 17.a, the designer has centered the final feedback in the first of the two radio buttons and has positioned and sized the interim feedback so that it appears to thicken the second of the two radio buttons. Lapidary knows that the constraints in the feedback objects need to be generalized so that the objects can highlight any of the items in the selectable set. The key to generalizing the constraints is two-fold: a) the constraints use pointers to reference the



(a) (b) (c)

Figure 18: (a) The various parts for a move behavior; (b) The parts, such as the selection handles, can be changed without affecting the move behavior; and (c) the dialog box for specifying the move/grow interactor. The attach-point indicates where the mouse should be attached to the object that is being moved or resized.

example objects; and b) the choice-of-items interactor stores a pointer to the object that should be highlighted in the feedback object's `obj_over` slot. Therefore, the constraints can be generalized by placing formulas in the pointer variables that retrieve the object to be highlighted from the `obj_over` slot. For example, if the feedback object has the constraints:

```
left = self->left_ptr->left - 2
left_ptr = circle-0301
```

Lapidary would set `left_ptr` to be equal to the object in `obj_over` (`left_ptr` = `obj_over`). If the selectable items are aggregates and the constraints in the feedback object reference a child in one of these aggregates, then Lapidary stores the path to the child in the pointer variable. In the above example, suppose circle-0301 is a child of the second radio button in Figure 17.a. Then Lapidary would store the formula `left_ptr = obj_over->circle` in `left_ptr`.

As in the case of object composition, Lapidary recognizes that it is more efficient to have the path directly encoded in the constraint at run-time, so when the feedback object is saved, the pointer variable (e.g., `left_ptr`) is replaced with the path (e.g., `obj_over->frame`) in the constraint that computes `left`. Lapidary is even smart enough to figure out that `obj_over` by itself, with no path, should be inserted into a constraint.

It should also be noted that sometimes multiple feedback objects are required because the items that the interactor operates on have different types. For example, the designer might want a move-grow interactor to be able to move line-like objects or box-like objects. In this case, the designer can create multiple feedback objects, select all of them, and hit the appropriate feedback button. Lapidary will then bring up C32 and prompt the designer for a constraint that determines when to use each kind of feedback object at run-time.

Objects-to-change must be generalized so that the interactor changes the appropriate run-time objects, rather than the example object the designer chooses. For example, in the move behavior shown in Figure 18, the designer wants the interactor to move the object highlighted by the selection handles, not the specific example object chosen for the `obj-to-change` parameter. Thus, Lapidary must figure out how to compute the object to change based on the start-where objects. To establish this relationship, Lapidary assumes that there will be a slot in either the start-where objects or their parent that points to the object to change. If such a slot exists, Lapidary creates a formula that retrieves the object to change from this pointer slot and stores the formula in the interactor's `obj-to-change` slot. If it cannot find such a slot, it will pop up a C32 formula window and ask the designer to type in a formula that explicitly defines the relationship. In the example shown in Figure 18, suppose that the selection-handle aggregate is the final feedback object for a choice-of-items interactor. Then the aggregate contains an `obj_over` slot that points to the example rectangle. Thus if the interactor's `obj-to-change` slot can access the aggregate's `obj_over` slot, it can figure out which object to move at run-time. The object that an interactor starts operating over is always stored in the interactor's `first_obj_over` slot, so the selection-handle that the interactor starts over at run-time will be stored in this slot. Since the aggregate is the parent of each of the selection handles, the object to be changed can be computed using the following formula:

```
obj-to-change = self->first_obj_over->parent->obj_over
```

7.3 Demonstration

Another way for the designer to specify how an interactor changes the picture is by demonstration. This is useful when objects in the picture should be modified, rather than highlighted using a *separate* object for feedback. For example, the currently selected item in a menu might be shown in italics (Figure 19), or the line-style of a circle in a radio button might be thickened (Figure 17).

Another use is to have buttons move to cover their shadows (and therefore look more ‘‘3-D’’), as is done for Lapidary’s own user interface (Figure 20). The conventional layered model, where the interim and final feedback are objects that are displayed on top of the selected items, does not work very well in these cases.

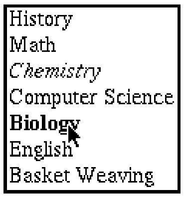


Figure 19: A menu where the the item under the mouse changes to bold and the final item selected is shown in italics.

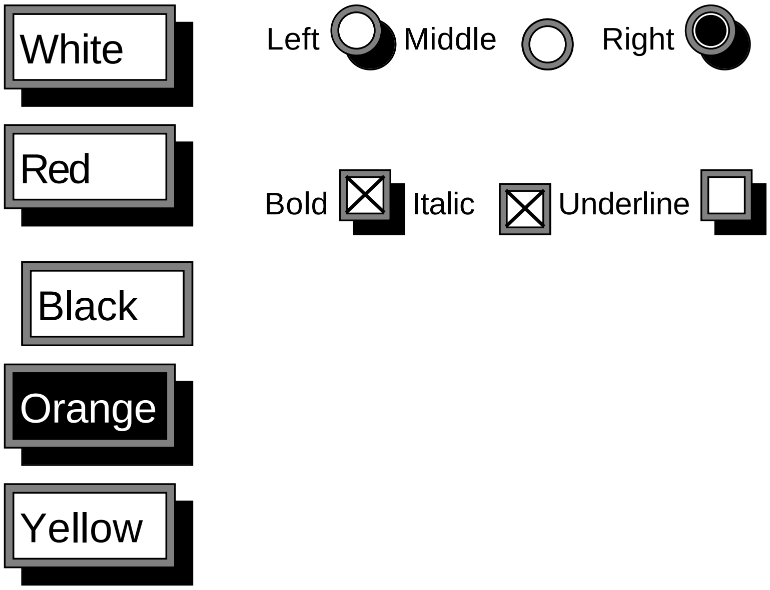


Figure 20: Three groups of buttons. The center button of each group has moved in ‘‘simulated 3 dimensions’’ (towards the shadow) to show that it is being pressed by the mouse (this serves as interim feedback). The reverse video rectangle, the black dot, and the X’s are final feedback objects.

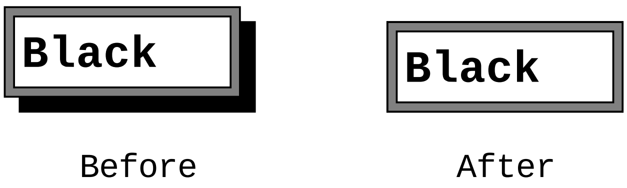
Lapidary supports a form of demonstrational programming [24] in which the designer specifies a ‘‘before’’ state for an example object, edits it, and presents Lapidary with an ‘‘after’’ state. Lapidary figures out the differences, determines how to implement the changes, and generalizes the behaviors so that it applies to any of the items that the interactor operates on.

To specify the changes by demonstration, the designer first selects the example object that will change, and then selects `By Demo` buttons in the choice-of-items dialog box (Figure 17). Lapidary creates a copy of the object. The designer then edits the object into the desired ‘‘after’’ state by using constraints and property menus.

When the designer is finished, Lapidary compares all the slots of the copy (the ‘‘before’’ state) with slots in the prototype (the ‘‘after’’ state). Two slots are considered different if: 1) they have different constraints; 2) they have no constraints but different values; or 3) one has a constraint and the other has no constraint. For each slot that differs, Lapidary synthesizes a constraint that selects between the two different values in the slots depending on the value of an indicator variable.

For example, suppose a designer wanted to demonstrate that a floating, 3D button should appear to depress when the mouse moves over it (Figure 20). This behavior could be demonstrated by selecting a floating button in the menu and pressing the `By Demo` button. Lapidary treats the example's button state as the ‘‘before’’ state. The designer could then select the gray frame of the button and change the constraints in the frame's `left` slot and `top` slots so that the frame is now positioned over the black shadow. The constraints that position the white frame and the label with respect to the gray frame automatically cause the white frame and label to move as well (Figure 21). The before and after states would look as follows:

	Before	After
`left:`	`self->left_ptr->left`	`self->left_ptr->left`
`left_ptr:`	`self->parent`	`self->parent->shadow`
`top:`	`self->top_ptr->top`	`self->top_ptr->top`
`top_ptr:`	`self->parent`	`self->parent->shadow`



(a) (b)

Figure 21: The ‘‘before’’ (a) and ‘‘after’’ (b) states of a 3D button. In (b) the designer has edited the gray frame so that it covers the black shadow. Constraints that connect the label and white frame to the gray frame have caused the white frame and label to stay properly aligned with the gray frame.

The ‘‘before’’ and ‘‘after’’ values of the `left_ptr` and `top_ptr` slots differ in this example, so Lapidary synthesizes constraints that choose between the differing values based on the value of an indicator variable. Since the choice-of-items interactor sets an object's `interim-selected` slot to true when the object should be highlighted, Lapidary chooses the `interim-selected` slot as the indicator variable. For example, the constraints might be:

```
left_ptr: if interim-selected then self->parent
                              else self->parent->shadow
top_ptr: if interim-selected then self->parent
                             else self->parent->shadow
```

In this example, only the values of slots were different. To illustrate what happens when constraints differ, suppose that the designer centered the gray frame over the black shadow instead of left-aligning it (since the size of the gray frame and the black shadow are the same, left-alignment and centering both position the gray frame over the black shadow—in practice we have found that people use both types of constraints to perform this positioning). In this case the before and after states would look as follows:

```
             Before                      After
left:        self->left_ptr->left        self->left_ptr->left +
                                           (self->left_ptr->width -
                                            self->width) / 2
left_ptr:    self->parent                self->parent->shadow

top:         self->top_ptr->top          self->top_ptr->top +
                                           (self->top_ptr->height -
                                            self->height) / 2
top_ptr:     self->parent                self->parent->shadow
```

Now, in addition to synthesizing constraints for `left_ptr` and `top_ptr`, Lapidary would also synthesize the following constraints for `left` and `top`:

```
left: if interim-selected
         then self->left_ptr->left +
              (self->left_ptr->width - self->width) / 2
         else self->left_ptr->left

top: if interim-selected
         then self->top_ptr->top +
              (self->top_ptr->height - self->height) / 2
         else self->top_ptr->top
```

Finally, Lapidary creates instances of these new constraints and installs them in the other buttons in the menu, thus generalizing the behavior from the example button to all items in the menu. Currently, demonstration in Lapidary is limited to the interim or final feedback of choice-of-items interactors. However, this feature could be easily extended to allow properties of objects that depend on ‘‘active values’’ [13] to be demonstrated this way, so that the graphics that change under application control can be demonstrated just as easily. For example, one could imagine demonstrating that a temperature monitor should change color when the temperature exceeds some threshold. In this case, the before state of the monitor would be one color, and the after state would be another color. One could even extend this paradigm so that multiple after states can be demonstrated, depending on different values or different ranges of the active value.

7.4 Testing and Saving Interactors

When the designer finishes filling in the interactor dialog box, the designer can create an instance of an interactor and immediately test it by putting Lapidary in ‘‘test’’ mode. The interactor will then behave exactly as if it were running in the application itself. If any changes need to be made, the designer can display the appropriate interactor dialog box, make the changes, and then press the modify button. The interactor can then be tested again, and the cycle repeated until the designer is satisfied with the design. At this point the interactor can be saved to a file and loaded into an application. Usually, however, Lapidary will propose an aggregate that the interactor should be attached to. Then, when the aggregate is saved, the interactor is automatically saved with the aggregate. Lapidary determines the aggregate by examining the objects specified by the `start-where` parameter and choosing the aggregate which is their parent. Lapidary asks the designer if is appropriate to add the interactor to this aggregate before actually adding it.

8 Example

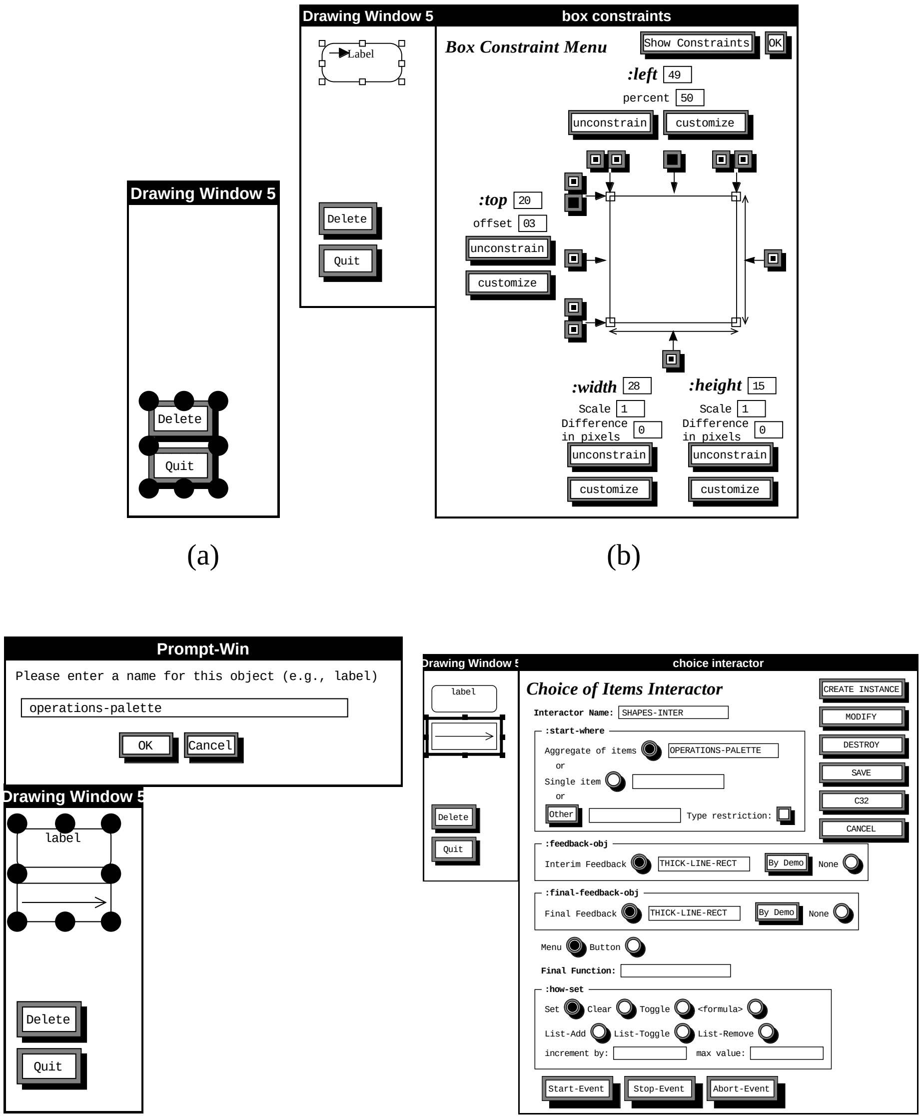
To illustrate Lapidary’s capabilities, suppose we wish to create the graphics and behavior for the boxes-and-arrows editor shown in Figure 2. The designer might divide this task into three parts: 1) creating the static presentation, which consists of the operations palette and delete/quit menu; 2) creating the application graphics and their constraints (e.g., the labeled boxes and arrows); and 3) defining the behaviors of these graphics (e.g., moving a box, editing a string). Although it takes a good deal of text to describe how to create this editor with Lapidary, the entire process of creating the graphics and their behavior takes only 15 minutes, and the callbacks can be coded in 25 minutes. This is much faster than what even the most experienced programmer could accomplish with a high level toolkit. For example, it would take at least 3 hours to code this application using the Garnet toolkit [21].

8.1 Static Presentation

The delete/quit menu is constructed by creating an instance of a Garnet text button widget and roughly positioning it. C32 is used to specify the names of the choices, `Delete` and `Quit`, and the application callback procedure (Figure 22.a).

The operations palette is built by creating the labeled box and arrow objects that comprise the palette and specifying a choice-of-items behavior. The labeled box is constructed by sweeping out a roundtangle, creating a text string called ‘‘label’’, centering the text string in the roundtangle using the iconic constraint menus, and creating an aggregate from the two objects (Figure 22.b). The arrow in the box is created similarly, and constrained to lie a fixed number of pixels below the bottom of the labeled-box object. Finally the arrow and labeled-box objects are bundled into an aggregate and named ‘‘operations-palette’’ (Figure 22.c).

Next the designer specifies the behavior for the operations palette. The user of the editor should be able to choose either the labeled box or arrow, so the designer creates a choice-of-items behavior. In Figure 22.d, the designer has brought up the choice-of-items dialog box and indicated that the operations-palette contains the set of items to be chosen from. To create the thick-lined rectangle that is used as both



(a) (b)

(c) (d)

Figure 22: Creating the static presentation of the boxes and arrows editor in Figure 2. (a) the designer creates and positions the delete/quit gadget. (b) The designer starts constructing the elements in the operations palette—the label for the labeled box object has just been centered in the frame. (c) The designer places the labeled box and arrow objects into an aggregate called ‘‘operations-palette’’. (d) The designer specifies that the choice-of-items interactor should start on either of the elements in ‘‘operations-palette’’ and that the thick-lined rectangle is the interim and final feedback.

interim and final feedback, the designer sweeps out a rectangle and selects a thicker line style from the line-style properties box. The feedback should be centered over the selected item and be slightly bigger than the selected item. To convey this information to Lapidary, the designer selects a sample menu item, in this case the arrow box, and uses Lapidary's iconic constraint menu to perform the necessary alignment and size adjustments. Finally the designer selects both the interim and final feedback buttons to indicate that this rectangle should be used as both interim and final feedback, names the interactor, and creates it (Figure 22.d). The designer finishes the static presentation of the editor by using constraints to left-align the delete/quit menu with the operations palette. The window can now be saved to a file.

8.2 Creating Application Graphics

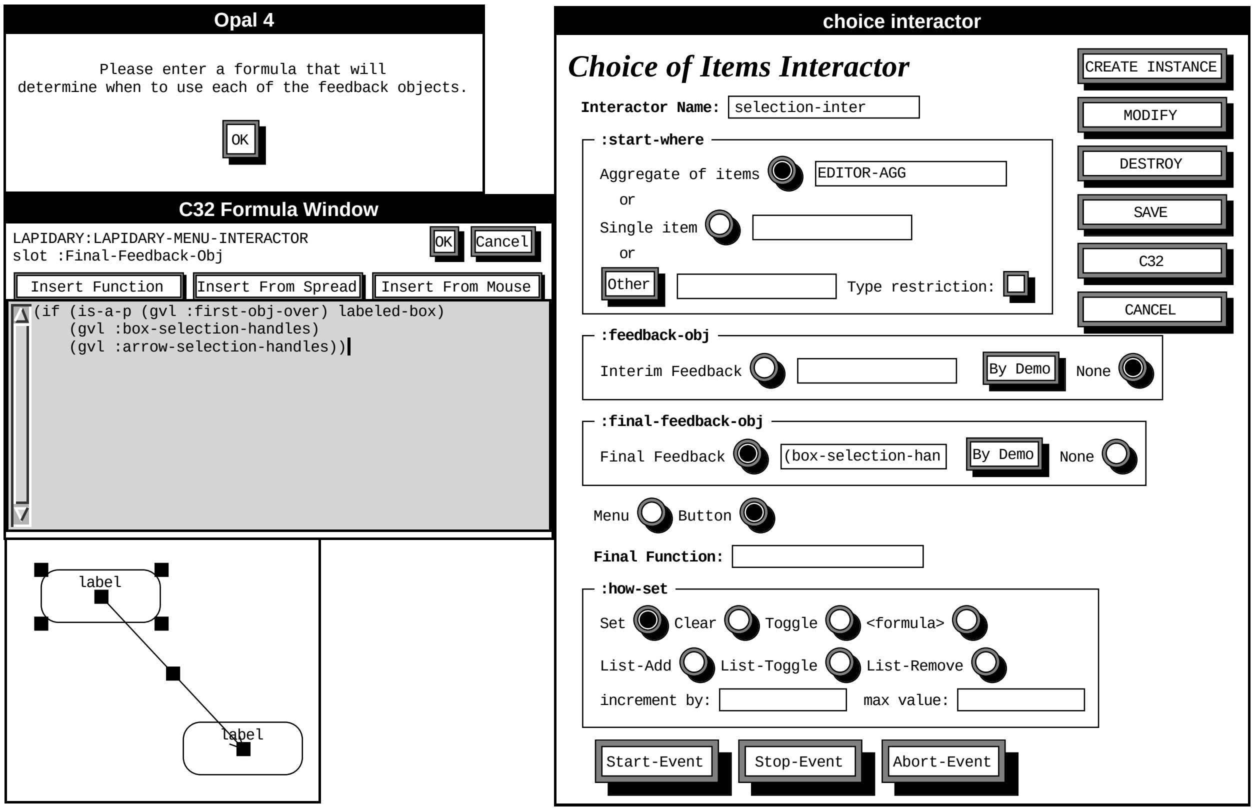
As shown in Figure 2, the boxes-and-arrows editor contains two types of application objects: labeled boxes and arrows. A prototype for the labeled box can be constructed by sweeping out a roundtangle and creating a text string. The designer can use Lapidary's iconic constraint menu to center the roundtangle around the label and make the roundtangle just large enough to contain the label. The use of constraints ensures that the roundtangle will expand and shrink as the label expands and shrinks. The label and the rounded rectangle are then grouped together in an aggregate, and the aggregate is named labeled-box. Finally the prototypical labeled-box is saved to a file so that the boxes-and-arrows editor can load it and create instances of it.

The designer creates the arrow prototype by selecting the arrow object from the shapes menu and drawing an arrow. In the boxes-and-arrows application, the arrow should be connected to the centers of two arbitrary boxes. To show this behavior, the designer creates two copies of the labeled-box object and positions them in the window. The endpoints of the arrow are then attached to the centers of the two boxes using Lapidary's line constraint menu. These constraints are dynamic constraints, since the arrow will stay attached to the data objects if the data objects are moved. When the arrow is saved to a file, Lapidary notices that the arrow's constraints reference objects that are not being saved. Lapidary infers that these constraints should be generalized so they can reference any object, and with the designer's permission, it parameterizes the references in these constraints.

8.3 Interactive Behavior

Finally, the designer needs to demonstrate the ways in which the graphical objects should respond to mouse and keyboard events. Suppose that the left mouse button can be used to select either labeled boxes or arrows. A labeled box should be highlighted by four selection boxes and an arrow by three selection boxes (Figure 23). The designer constructs the selection handles for boxes by creating four rectangles, positioning them using constraints over the four corners of an example labeled-box, and making an aggregate out of them. Similarly, the designer constructs the selection handles for arrows by creating three rectangles, positioning them using constraints along an example arrow, and making an aggregate out of them.

To specify the selection behavior the designer needs to create a choice-of-items behavior. The designer would like the choice interactor to be able to select any box or arrow object in the boxes-and-arrows editor. However, since the editor is not running, those objects are not created yet. Therefore the designer must demonstrate the behavior on example objects. To do this, the designer creates an aggregate consisting of the example arrow and the two labeled boxes that were created in the previous section.



(c) (d)

Figure 23: The creation of a selection behavior. Arrows should be highlighted using three selection handles and labeled boxes should be highlighted using four selection handles. Lapidary is currently prompting the designer to enter a formula that determines when to use each of the selection handles. Window (a) is the prompt window, window (b) is the window for entering the formula, window (c) is the drawing window that contains the examples of selectable objects and the selection handles that should highlight them, and window (d) is the dialog box that is being used to create the selection behavior.

Using the choice-of-items dialog box, the designer indicates that the set of objects in this aggregate is the set of objects that the interactor can select. The designer then selects the aggregate of four squares that is the feedback for labeled boxes, and the aggregate of three squares that is the feedback for arrows. After the designer hits the final feedback button in the choice-of-items dialog box, Lapidary pops up C32 and a message box asking for a constraint that will determine when to use each of the two feedback objects. The designer creates a constraint that looks at the type of the selected object and returns the appropriate feedback object. The designer then names the interactor and creates it. Lapidary asks the designer if the interactor should be attached to the aggregate of all created objects, and the designer gives it permission to do so.

Next, the designer creates a move behavior that moves a selected labeled box when the left mouse button is pushed down over one of the selection boxes that highlight the labeled box. Section 7.2 explains how this behavior can be specified and Figure 18 shows the dialog box that specifies it. Notice that the move behavior can start only when it is over a labeled box's selection handles. Since the selection handles for arrows are not start-where objects, arrows cannot be moved by this interactor.

In a similar fashion, the designer could create interactors for editing the labels in the labeled boxes and creating new objects. The designer could then save the aggregate containing these interactors and use the aggregate in the boxes-and-arrows editor at run-time. To complete the application, the designer would write a callback procedure for the delete/quit menu and a callback procedure for the new object interactor that created the appropriate object. Our future research will be investigating how to allow these callbacks to be demonstrated visually rather than programmed textually [23].

9 Practice and Experience

9.1 User Experience

Lapidary has been released as part of the Garnet user interface toolkit for approximately a year. Before its release it was extensively tested on students in various user interface courses at the University of Tennessee. Users report a somewhat steep initial learning curve. However, once they are familiar with Lapidary, they find it easier to create user interfaces using Lapidary than using the programming layers of Garnet. A rough measure of the difference between Lapidary and the programming layers of Garnet is that students generally took ten or more weeks to built a user interface in Garnet and roughly two to three weeks to built the same interface in Lapidary. However, the students were concurrently taking other courses, so we do not have precise elapsed time data. In general users can create virtually all of the application graphics and behaviors using Lapidary (Gilt is preferred for creating widgets and Lapidary does not yet support less frequently created shapes such as polygons and arcs).

Users report greater difficulty learning how to create behaviors than graphics in Lapidary. This appears to be a result of their trying to do more than Lapidary can support. Lapidary is only intended to support the ‘‘syntactic’’ aspects of a user interface, not the ‘‘semantic’’ aspects. The ‘‘syntactic’’ aspect is the response of graphics to user input and application input. This includes the handling of input events and the formatting of application data. The ‘‘semantic’’ aspect refers to the callback procedures that should modify the application data structures. These procedures might create or delete objects, determine which boxes should be connected to the endpoints of arrows, or update the values in an arithmetic editor. We have found that users attempt to specify both syntactic and semantic actions with Lapidary, and become confused when they have difficulty specifying semantic actions. Once users understand what Lapidary is and is not capable of doing, their productivity improves greatly. A number of students on whom Lapidary was tested remarked afterwards that it seems so simple and were amazed that it took them as long as it did to use it effectively. A new tool that is currently being developed as part of Garnet, called Marquise [23], is intended to provide support for specifying semantic actions in an application.

9.2 Implementation Experience

Because the graphics and behavior model of Garnet is so well-defined, it was quite easy to implement Lapidary's drawing editor, behavior-defining dialog boxes, and inferencing capabilities [22]. Throughout this paper we have described how various features of Garnet's prototype/instance, constraint, and interactor systems helped to simply the implementation of Lapidary. However, one area in which we did encounter a great deal of difficulty was in handling aggregates.

Conceptually aggregates are quite simple—they are collections of objects. However, constraints are extensively used to lay out the parts of the aggregate and to transmit parameters from the top-level aggregate to the children. Indeed, except for the position and size slots that contain the information to be propagated around, all the other position and size slots in the aggregate and its children need constraints. To see that this is so, suppose that a programmer created a labeled box in which the position information flowed from the top-level aggregate to the children. If the programmer forgets to place constraints in the children that retrieve the position information from the parent, then the children will not move when the position information in the top-level aggregate changes. Similarly if size constraints are not placed in the top-level aggregate, the top-level aggregate will not expand or shrink when the label expands or shrinks.

When programmers create aggregates textually, they explicitly specify all necessary constraints, and they can defer displaying the aggregates until all necessary constraints have been created. However, this is tedious, and in practice, getting all these constraints right the first time without using a design tool is difficult. Typically an aggregate must be debugged a number of times in order to get its appearance correct.

When aggregates are created graphically, it is not possible to defer displaying them until all their constraints have been specified, since the aggregate must be both created and selectable before constraints can be added that relate the aggregate to its children. One alternative is to simply display the aggregate and its children and require the designer to explicitly specify all the constraints, as is done when the designer creates the aggregate textually. This is less error-prone than the textual case since the designer can immediately see the impact of each constraint. However, it is still tedious, and if the designer forgets even one constraint, the aggregate will not move or resize correctly. Thus Lapidary tries to infer the constraints.

The problem with inferring constraints is trying to decide whether an aggregate is a ''top-down'' aggregate or a ''bottom-up'' aggregate. In a top-down aggregate, position and size information flow from the aggregate to its children, whereas in a bottom-up aggregate, this information flows from the children to the aggregate. The inferencing process is further complicated by the fact that size and position can be orthogonal (i.e., specified independently), and thus size information might flow in one direction and position information in another direction. A good example of this is in a labeled box where the size of the labeled box may depend on the size of the label, while the position of the labeled box depends on the position of the top-level aggregate. In this case, size information flows from the children to the parent, and the position information flows from the parent to the children.

When a programmer writes or applies the constraints explicitly, the information flow is explicitly specified in the constraints. However, Lapidary has to guess which slots contain information to be propagated, and which slots should contain constraints that retrieve this information. In the applications we have worked with, the aggregates most often contain the position information (i.e., top-down) and the children most often contain the size information (i.e., bottom-up). Thus Lapidary places constraints in the children that compute the children’s position from the aggregate’s position, and places constraints in the aggregate that computes the aggregate’s size from the childrens’ size. However, Lapidary does not modify any slots which already have constraints.

A designer is free to override Lapidary’s guesses using the constraint menus. For example, the sizes of the components in a button typically depend on the size of the button. However, Lapidary will initially place constraints in the button aggregate that computes the button’s size based on the size of the button’s components. Thus the designer must change the constraints so that the children’s sizes depend on the button’s size. The problem is that designers sometimes forget to make these overrides, because when the aggregate is created, all the position and size information appears to be computed correctly (the aggregate’s bounding box is always just big enough to surround the aggregate’s components). If the designer then saves the button to a file, and at run-time, tries to set the size of the button to a different value, nothing will happen—the sizes of the button’s components do not depend on the size of the button, so the components will remain the same size. Thus Lapidary cannot guarantee that the designer will always create the desired result, and the designer may have to do some debugging. However, if the designer does some preliminary testing of the button in Lapidary (e.g., tries to change the size of the button), it will quickly become clear that the button is not resizing properly, and the designer can easily fix the problem. Thus, although Lapidary cannot prevent aggregates from being incorrectly specified, its environment encourages a designer to experiment with a design, and provides facilities for easily fixing any problems that are discovered.

10 Future Directions

One of the limitations of Lapidary is that it is not able to specify the semantic actions of applications that are found in callback procedures. For example, it cannot specify how an object should be deleted from an application, and what the side-effects of that deletion are (e.g., deleting the arrows attached to a box). A new Garnet tool, Marquise, is currently being developed which explores how semantic actions might be specified interactively [23].

Within Lapidary, we will be looking to minimize the need for dialog boxes by making more inferences from demonstration. For example, if the designer moves a mouse back and forth over a list of objects, Lapidary might guess that a menu interactor is desired without requiring any additional specification. If there are nearby objects that resemble feedback objects (e.g., they are Xored to the screen), Lapidary may infer that these objects are feedback objects and after a few mouse motions, attach one of these objects to the mouse. Of course, if Lapidary makes the wrong guesses, the designer can always correct them using the dialog boxes.

We would also like to enhance the drawing editor’s ability to specify the lay out the application’s output. Lapidary currently provides the ability to organize the application’s output into lists, but it would be useful if other representations, such as tables, trees, and graphs were provided. Garnet provides formatted trees and graphs that provide a high degree of customization through parameters. It would be also interesting to add an inferencing mechanism that based on a user's manipulation with example objects in the output structures, attempted to guess which parameters were to be changed. Since Garnet's list, tree, and graph structures all provide a well-defined set of parameters, the inferencer would have only a limited set of options to choose from, and thus its chances of guessing correctly should be quite good.

11 Conclusions

Through the use of a direct-manipulation graphical editor, constraints which can be automatically generalized, and a small number of primitive interactive behaviors, Lapidary allows the user interface designer to create widgets and run-time application objects with either a custom or a standard ‘‘look and feel.’’ The behavior can be specified separately from the graphics by selecting among a small number of options using dialog boxes and by demonstration. If the designer requires more complex behavior, specialized constraints can be written using a spreadsheet-like interface and attached to any property of an object. The use of graphical, direct manipulation techniques to specify the graphics and constraints, and the use of programming-by-demonstration to specify the behavior, makes the creation of application objects significantly easier. Although there are only a small number of built-in constraints and interactors, these are able to cover a wide range of user interfaces, including most forms of widgets and many kinds of application-specific interactions. Lapidary is unique among existing interactive interface design tools in that it allows application graphics to be built from scratch, in that it provides a behavioral model that is independent of the graphics and that allows the graphical objects' response to input events to be specified pictorially, and in that it allows the dynamic layout of objects to be specified pictorially via constraints. Since creating these application-dependent aspects of an application takes about two orders of magnitude more effort than the creation of the application-independent aspects, Lapidary represents a significant advance on existing interface builders.

Acknowledgements

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. Additional support for the first author was provided by the Hewlett Packard Corporation.

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.

We wish to thank Roger Dannenberg, Dave Kosbie, and David Bolt who helped implement parts of Lapidary and participated in preliminary design discussions. We would also like to thank Dario Giuse and Andrew Mickish, who participated in the preliminary design discussions as well.

References

1. Alan Borning. Thinglab--A Constraint-Oriented Simulation Laboratory. Tech. Rept. SSL-79-3, Xerox Palo Alto Research Center, July, 1979.

2. W. Buxton, M.R. Lamb, D. Sherman, and K.C. Smith. Towards a Comprehensive User Interface Management System. Computer Graphics, Proceedings SIGGRAPH'83, Detroit, Mich, July, 1983, pp. 35-42.

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. Allen Cypher. EAGER: Programming Repetitive Tasks by Example. Human Factors in Computing Systems, Proceedings SIGCHI'91, New Orleans, LA, April, 1991, pp. 33-39.

5. Alessandro Giacalone. XY-WINS; An Integrated Environment for Developing Graphical User Interfaces. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'88, Banff, Alberta, Canada, Oct., 1988, pp. 129-143.

6. D. Austin Henderson, Jr. The Trillium User Interface Design Environment. Human Factors in Computing Systems, Proceedings SIGCHI'86, Boston, MA, April, 1986, pp. 221-227.

7. 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.

8. Tyson R. Henry, Scott E. Hudson and Gary L. Newell. Integrating Gesture and Snapping into a User Interface Toolkit. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'90, Snowbird, Utah, Oct., 1990, pp. 112-121.

9. Solange Karsenty, James A. Landay, and Chris Weikart. Inferring Graphical Constraints with Rockit. Human Computer Interaction: Proceedings of HCI'92, York, United Kingdom, Sept, 1992.

10. Henry Lieberman. "Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems". *Sigplan Notices 21*, 11 (Nov. 1986), 214-223. ACM Conference on Object-Oriented Programming; Systems Languages and Applications; OOPSLA'86.

11. David L. Maulsby and Ian H. Witten. Inducing Procedures in a Direct-Manipulation Environment. Human Factors in Computing Systems, Proceedings SIGCHI'89, Austin, TX, April, 1989, pp. 57-62.

12. David L. Maulsby, Ian H. Witten, and Kenneth A. Kittlitz. Metamouse: Specifying Graphical Procedures by Example. Computer Graphics, Proceedings SIGGRAPH'89, Boston, MA, July, 1989, pp. 127-136.

13. Brad A. Myers. "Creating Interaction Techniques by Demonstration". *IEEE Computer Graphics and Applications 7*, 9 (Sept. 1987), 51-60.

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

15. Brad A. Myers. "User Interface Tools: Introduction and Survey". *IEEE Software 6*, 1 (Jan. 1989), 15-23.

16. Brad A. Myers. Encapsulating Interactive Behaviors. Human Factors in Computing Systems, Proceedings SIGCHI'89, Austin, TX, April, 1989, pp. 319-324.

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

18. Brad A. Myers. "A New Model for Handling Input". *ACM Transactions on Information Systems 8*, 3 (July 1990), 289-320.

19. 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.

20. Brad A. Myers. Separating Application Code from Toolkits: Eliminating the Spaghetti of Call-Backs. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'91, Hilton Head, SC, Nov., 1991, pp. 211-220.

21. Brad A. Myers, Dario Giuse, Roger B. Dannenberg, Brad Vander Zanden, David Kosbie, Philippe Marchal, Ed Pervin, Andrew Mickish, John A. Kolojejchick. The Garnet Toolkit Reference Manuals: Support for Highly-Interactive, Graphical User Interfaces in Lisp. Tech. Rept. CMU-CS-90-117-R, Carnegie Mellon University Computer Science Department, August, 1992.

22. Brad A. Myers and Brad Vander Zanden. "An Environment for Rapid Creation of Interactive Design Tools". *The Visual Computer; International Journal of Computer Graphics 8*, 3 (1992), 94-116.

23. Brad A. Myers, Richard G. McDaniel, and David S. Kosbie. Marquise: Creating Complete User Interfaces by Demonstration. Human Factors in Computing Systems, Proceedings INTERCHI'93, Amsterdam, The Netherlands, April, 1993, pp. 293-300.

24. Brad A. Myers. Garnet: Uses of Demonstrational Techniques. In Allen Cypher, Ed., *Watch What I Do: Programming by Demonstration*, MIT Press, Cambridge, MA, 1993, pp. 219-236.

25. Frances J. Newbery. An interface description language for graph editors. 1988 IEEE Workshop on Visual Languages, Pittsburgh, PA, Oct., 1988, pp. 144-149. IEEE Computer Society Order Number 876.

26. Dan R. Olsen, Jr. "Larger Issues in User Interface Management". *Computer Graphics 21*, 2 (April 1987), 134-137.

27. Dan R. Olsen. "Editing Templates: A User Interface Generation Tool". *IEEE Computer Graphics and Applications 6*, 11 (Nov. 1986), 40-45.

28. Gurminder Singh and Mark Green. Designing the Interface Designer's Interface. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'88, Banff, Alberta, Canada, Oct., 1988, pp. 109-116.

29. 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.

30. Brad Vander Zanden, Brad A. Myers, Dario Giuse and Pedro Szekely. The Importance of Pointer Variables in Constraint Models. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'91, Hilton Head, SC, Nov., 1991, pp. 155-164.

31. John M. Vlissides and Steven Tang. A Unidraw-Based User Interface Builder. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'91, Hilton Head, SC, Nov., 1991, pp. 201-210.

32. David Wolber and Gene Fisher. A Demonstrational Technique for Developing Interfaces with Dynamically Created Objects. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'91, Hilton Head, SC, Nov., 1991, pp. 221-230.

Table of Contents

List of Figures