

Garnet


Comprehensive Support for Graphical, Highly Interactive User Interfaces

Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Vander Zanden,
David S. Kosbie, Edward Pervin, Andrew Mickish, and Philippe Marchal
Carnegie Mellon University

User interface software is difficult and expensive to implement.¹ Highly interactive interfaces are among the hardest to create, since they must handle at least two asynchronous input devices (such as a mouse and keyboard), real-time feedback, multiple windows, and elaborate, dynamic graphics.

The Garnet research project is creating a set of tools to aid the design and implementation of highly interactive, graphical, direct manipulation user interfaces. Garnet also helps designers rapidly prototype different interfaces and explore various user interface metaphors during early product design.

Most graphical interfaces are created using toolkits, collections of interaction techniques (sometimes called “widgets” or “gadgets”) such as menus, scroll bars, and buttons. Examples include the Macintosh Toolbox and Xtk for the X Window System. Unfortunately, these toolkits are often difficult to use, since they contain literally hundreds of procedures. Also, many toolkits do not help the programmer create the most important part of the application — the graphics that appear in the main application window. In particular, the application must handle all input events (expressed at a low level as “the left mouse button went down,” “the mouse is at (30,345),” etc.), deciding which operation to perform on objects and drawing objects with the



Garnet helps create highly interactive user interfaces by emphasizing easy specification of object behavior, often by demonstration and without programming.

underlying graphics package (which usually supplies operations such as “draw-line” and “draw-circle”). Furthermore, modifying or creating toolkit items is usually difficult or impossible.

Higher level tools such as interface builders and user interface management systems¹ have not adequately addressed these problems. A conventional interface builder lets a designer graphically place user interface components in a window, thereby creating menus, palettes, and dialogue boxes. Examples include Next’s In-

terface Builder, Smethers Barnes’ Prototyper for the Macintosh, and UIMX for X Windows. These programs let a designer place only preprogrammed interaction techniques in windows, and they usually allow only a few parameters to be set. Often, a designer will type the name of a procedure to be called when the interaction technique is executed. Designers cannot modify or design the interaction techniques themselves, and the interface builders do not address application-specific graphics at all.

A user interface management system helps a designer handle the dialogue, or sequencing, aspects of the user interface — that is, what happens after each user action. Such systems have generally not addressed the creation of toolkit components (menus, scroll bars, etc.) or the specification or management of application-specific graphical objects (the contents of application windows).

A number of features differentiate Garnet from other user interface tools, including an emphasis on handling objects’ runtime behavior (how they change when the user operates on them) and on handling all visual aspects of a program’s user interface, including its graphics and the contents of all application-specific windows.

For example, when creating an application that lets the user manipulate boxes connected by arrows (such as a graph edi-

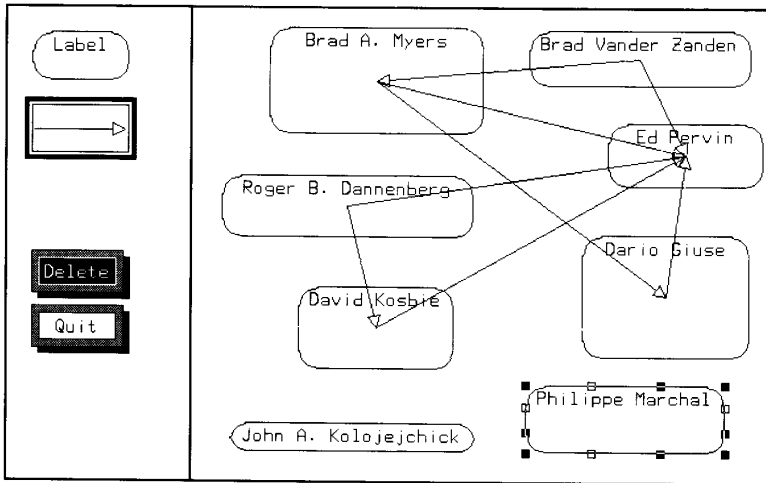


Figure 1. A simple boxes-and-arrows editor created with Garnet. An arbitrary number of new boxes and arrows can be added, and their initial position is specified using the mouse. Any existing box or arrow can also be selected, changed in position or size, and deleted. This entire editor was implemented in about three hours using the Garnet Toolkit.

tor or project-planning chart — see Figure 1), Garnet's user interface builder lets the designer draw the boxes and arrows and demonstrate how they should respond to the mouse. Menus, palettes, and dialogue boxes can be generated automatically or added graphically. Toolkits and interface

builders in other systems might help build the menus and palettes, but they don't let the designer implement and manage the boxes and arrows themselves. Designers using those programs must code these elements directly using the underlying graphics package and input device handlers.

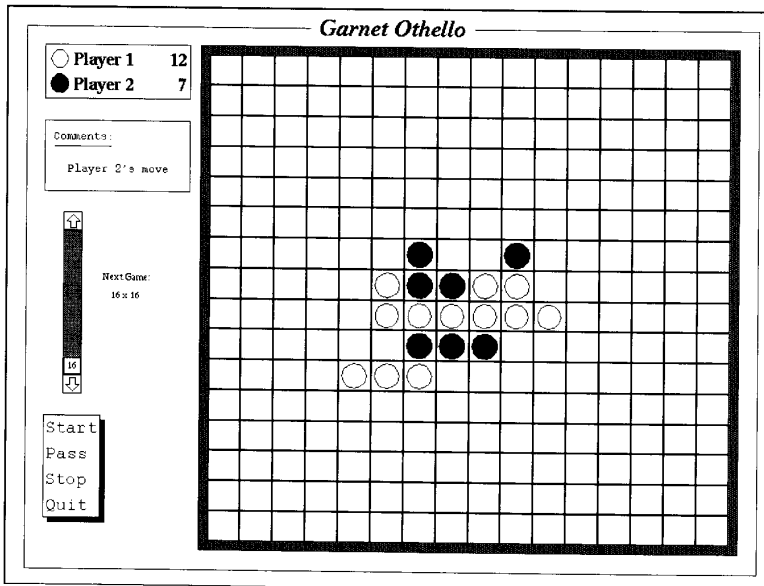


Figure 2. An Othello game created with Garnet.

Typically, coding application-specific graphics requires 10-100 times more effort than dealing with menus and buttons, so Garnet should save programmers significant time.

This article provides an overview of all parts of Garnet, which stands for "generating an amalgam of real-time novel editors and toolkits." Previous articles²⁻⁵ have concentrated on individual aspects of the system. (There is also a complete reference manual for the Garnet Toolkit.⁶)

Garnet is entirely "look-and-feel independent," which means the designer can either create user interfaces with an original graphical appearance and behavior or choose from a set of predefined appearances and behaviors.

The system contains both low-level and high-level tools. The low-level tools, called the Garnet Toolkit, use a number of mechanisms to help implement user interfaces, including a prototype-instance object system, a constraint system (which allows relationships among objects to be declared once and then maintained by the system), automatic graphical object updating (so that the system refreshes the display when objects change), and separate specification of input handling from graphics.

Garnet's high-level tools include the Lapidary interface builder; the Jade dialogue box creation system, which automatically creates menus and dialogue boxes from a list of their contents; and the C32 spreadsheet for specifying complex constraints. Garnet is implemented in Common Lisp and uses the X Window System through the standard CLX interface from Common Lisp to X Windows. A version for the Macintosh is also being developed. Garnet is therefore portable and runs (so far) on CMU, Lucid, Allegro, Harlequin, and TI Common Lisps and on many hardware platforms. Garnet does not use the Common Lisp Object System (CLOS) or any Lisp or X Windows toolkit (such as Interviews,⁷ CLUE, CLIM, or Xtk).

Coverage

Garnet is designed to handle user interfaces that let users operate on graphic objects with the mouse and keyboard. Since the objects often represent application data, and since changes made on the screen to the graphics are translated into changes to the data, the user has the feeling he or she is directly manipulating the data. Similarly, the screen reflects changes the application makes to the data, possibly in response

to external events.

Garnet is suitable for

- box and arrow diagram editors (see Figure 1), like MacProject;
- conventional drawing programs, such as MacDraw;
- icon manipulation programs, like the Macintosh Finder;
- graphical programming languages in which computer programs are constructed using icons and other pictures, such as a flowchart;
- tree and graph editing programs, including semantic networks, neural networks, and state transition diagrams;
- board games, such as chess or Othello (see Figure 2);
- simulation and process monitoring programs, in which the user interface shows the status of the monitored simulation or process and lets the user manipulate it;
- user interface construction tools (we implemented Garnet using itself); and
- some forms of CAD/CAM programs.

Garnet does not have a significant text editing component, but it does provide small editable strings that might be used as labels or fields in a table or dialogue box.

Garnet's primary influence is Peridot,⁸ a construction tool that lets users create toolkit items without programming. Peridot lets nonprogrammers create many types of interaction techniques, including most kinds of menus, property sheets, buttons, scroll bars, percent-done progress indicators, sliders, and iconic and title line window controls. Like Garnet, Peridot uses constraints. However, Peridot lacks a programming interface and offers no way to use existing toolkit items or create application-specific graphic objects. (Many systems, including Thinglab⁹ and Apogee,¹⁰ have used constraints; see the sidebar.)

Garnet's Jade dialogue editor, which automatically constructs dialogue boxes from high-level specifications, was influenced by the Interactive Transaction System.¹¹

The Garnet Toolkit

Garnet contains a number of different components grouped into two layers. The Garnet Toolkit (the lower layer) supplies the object-oriented graphics system and constraints, a set of techniques for specifying the objects' interactive behavior in response to the input devices, and a collection of interaction techniques. Designers using this layer must write procedures. The

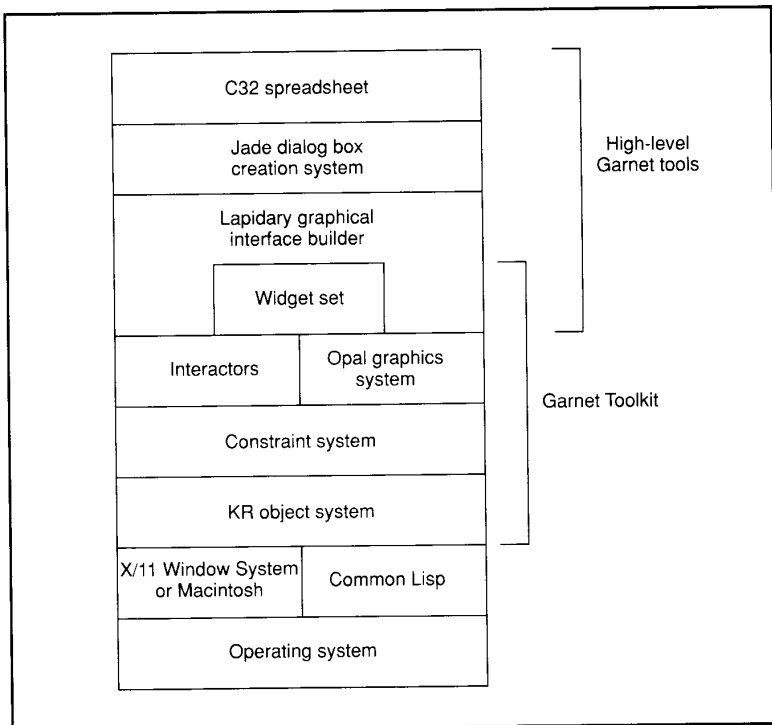


Figure 3. The structure of the Garnet system.

higher layer contains tools that let designers draw pictures to show how the interface should look and behave and to define parts of the interface at a high level. These tools automatically create code based on the user's specifications.

The toolkit itself is divided into several components (see Figure 3):

- an object-oriented programming system,
- a constraint system,
- a graphical object system,
- a system for handling input, and
- a collection of gadgets or widgets.

Using the X toolkit's terminology, the first four parts of the Garnet Toolkit are intrinsics (the mechanisms supporting the implementation) and the fifth is the widget set (a collection of menus, scroll bars, etc., with a prespecified look and feel).

Object-oriented programming system.

Garnet's object-oriented programming system, called KR (for "knowledge representation"), supports a prototype-instance model for objects rather than the conventional class-instance model used by Small-

talk and C++. A prototype-instance model does not distinguish between instances and classes; any instance can be a "prototype" for other instances. All data and methods are stored in "slots" (sometimes called fields or instance variables). Slots that are not overridden by a particular instance inherit their values from their prototypes. Actually, KR does not distinguish between data and method slots. Any slot can hold any type of value, and in Common Lisp a function is just a type of value. Slot names start with colons and can contain any number of printing characters (for example, :left, :interim-selected, :obj-over).

An instance can also add any number of new slots. Unlike conventional class-instance models, this means that the number of slots in each object is highly variable — each object can have a different number of local slots, depending on which properties it wants to accept as defaults and which it wants to override. In fact, the number of slots can change dynamically. Slots can be explicitly removed from objects at any time. If a program sets the value of a slot that does not exist, then the slot is created automatically. The advantage of this fea-

ture is that it is easy to create slots to hold local data. For example, if a rectangle's color represents an application temperature, the application can create an instance of a rectangle and a slot in it called :temperature. A disadvantage of this flexibility is that compile-time or even runtime checking of slot accessing and setting is impossible, since all slot names are legal. Also, since slots can hold any type of value, there is no type checking at the KR level.

As an example of inheritance, assume that a top-level rectangle prototype has

slots containing values for the rectangle's left, top, width, height, and color (see Figure 4). An instance using that rectangle as a prototype will typically — but not necessarily — override some of the values. A programmer could create an instance with a particular color and size, and then create more instances of that instance (see Figure 4). If a prototype's slot value changes, all instances of that prototype that do not override that slot immediately inherit the new value. Similarly, if an instance's slot is removed, the corresponding slot of its

prototype will be used instead, if it exists. All objects (prototypes and instances) can be displayed on the screen.

The methods that implement messages sent to the objects can also change dynamically, which is not possible in conventional object systems like Smalltalk. In Garnet, the designer need only assign a new procedure to the appropriate slot; the new method will be used subsequently.

The prototype-instance model is more dynamic and flexible than the familiar class-instance model. A high-level tool such as

A constraints primer

Brad Vander Zanden and Brad A. Myers

Any large, complex application contains thousands of interdependent relationships. For example, a graphical application must deal with the relationships arising from laying out objects, displaying feedback for input operations, and keeping views consistent with the underlying data they represent. These relationships might include keeping text labels centered in boxes, making feedback objects follow the cursor during a move operation, and setting the color of an airplane icon according to whether a flight is early, on time, or delayed.

Constraints provide a convenient way to specify relationships and have them automatically maintained at runtime by a constraint solver. In contrast, a conventional programming language requires the application to both specify the relationships and do all the bookkeeping to maintain them. For example, suppose that an arrow must stay connected to the center of a box and that a label on the arrow must stay centered on the arrow. In constraint programming, whenever the application changes the box's position or size, the constraint solver automatically repositions the arrow and its label. In contrast, a conventional language forces the designer to write code to reposition the arrow and the label. This might not seem too onerous a task, but when an application contains thousands of such relationships, the bookkeeping needed to maintain them increases so rapidly that adding new functionality to the application becomes difficult, and the time required to debug the changes also increases substantially.

In addition to simplifying the creation of an application and increasing its robustness, constraints lend themselves to incremental recomputation. When a user changes one or more parameters in an application, or adds or deletes a number of constraints, most of the existing constraints remain satisfied and only a small number must be reevaluated. An incremental constraint-solving algorithm can automatically identify which constraints must be reevaluated and limit its solving to these constraints. Such an algorithm can be used with any application written for that constraint system. In contrast, a conventional language requires the designer to create a new incremental algorithm for each application. Of course, a conventional language also lets the designer take advantage of any special characteristics of the application, so that he or

she can write a custom algorithm that might be faster than a general-purpose constraint solver.

Types of constraints

Constraints can be either one-way or multiway. One-way constraints let the constraint solver change only one object in the constraint to satisfy it; multiway constraints allow any object to change. For example, the arrow in Figure 5 (in the main text) is connected to the circle and the box by one-way constraints. If either the box or the circle moves, the arrow also moves to resatisfy the constraints. However, if the arrow moves, neither the box nor the circle moves — the constraints that tie the arrow to the box or circle are violated or removed. If these constraints were multiway, then the constraint solver would move the box or circle when the arrow moved, thus satisfying the constraints.

Multiway constraints are obviously more powerful than one-way constraints, but this increased expressiveness comes at a price. One-way constraint solving algorithms only have to evaluate constraints, making them simpler to implement and more efficient than multiway constraint solvers that must also choose which variable in a constraint should be modified.

Multiway constraints can also introduce ambiguity at the design level. For example, suppose we have the constraint $A - B - C = 0$. If A changes, the constraint solver must choose whether to change B , C , or both. In this case, we should adhere to the principle of least astonishment — a user's editing operation should change the result in a way consistent with the user's expectations.^{1,2} One approach to eliminate this ambiguity — constraint hierarchies² — divides constraints into hierarchies or priority levels. The constraint solver then tries to satisfy as many constraints as possible, solving the highest priority constraints first, then the next highest, and so on. One issue that constraint hierarchies do not address is how to specify that multiple values should change (for example, when both B and C should change when A changes).

Constraint solving

Constraints can be solved in either a lazy or eager fashion. Lazy evaluation evaluates a constraint only if it affects a result

the Garnet interface builder can display a prototype on the screen and let the user edit it. All instances of that prototype automatically reflect these edits. For example, if a designer changes the standard look and feel of a menu prototype, all menus in the system immediately change accordingly. In most class-instance systems, changing the class structure when instances exist is either very expensive or makes the instances invalid.

A potential disadvantage of the prototype-instance model is that getting a slot's

value could require a search up the entire inheritance hierarchy, since slots must be inherited from the prototypes if they are not present in an object. Garnet alleviates this problem by keeping local caches of inherited values. Another efficiency problem is that, since new slots can be added at any time, a fixed storage scheme like a structure or record cannot be used. Rather, a dynamic list of slot names and slot values must be kept. Garnet addresses this problem by keeping the most commonly accessed slots in a Lisp record structure so

they can be accessed quickly, and by keeping any new slots in a list.

The efficiency of Garnet's object system is actually better than other Lisp object systems, such as CLOS. For example, on a Sun 3/60 workstation running Allegro Common Lisp, the simplest slot accessing function takes 49.8 microseconds in Portable Common Loops (an implementation of CLOS), but only 27.3 microseconds in KR. Creating an instance in CLOS (16,160 microseconds) takes about 15 times longer than in Garnet (1,117 microseconds).

that the user requests; eager evaluation evaluates a constraint immediately when values change. Thus, a lazy-evaluation system can contain variables whose values are out of date. Lazy evaluation avoids unnecessary work if relatively few values are needed to compute the result the user requests. For example, if portions of a drawing are off screen, they might not have to be recalculated. However, lazy evaluation also introduces extra bookkeeping, since the constraint solver must keep track of out-of-date variables. In addition, lazy evaluation can result in potential delays when the values of out-of-date variables are demanded. Lazy evaluation is most effective in applications where the user wants to view only a limited portion of the application's data and where changes are occurring to all parts of the application's data. Otherwise, eager evaluation is preferable, since it is conceptually cleaner than lazy evaluation (everything is always up to date).

Constraint systems

Examples of graphical systems that use constraint technology abound.³ Sketchpad⁴ pioneered the use of graphical constraints in a drawing editor in the early 1960s. Thinglab¹ took constraints a step further and introduced them into the realm of graphical simulation. More recently, Thinglab has been refined to aid in the generation of user interfaces.² Both Sketchpad and Thinglab provide multiway constraints. However, most user interface development systems use one-way constraints because of their simplicity and efficiency; Grow,⁵ Peridot,⁶ and Apogee⁷ are three examples. Grow was the first comprehensive user interface development system to use constraints. Peridot was the first to try to infer constraints. Apogee was the first to employ lazy evaluation. Constraint⁸ provided a user interface development environment that used multiway constraints and introduced a new constraint-solving algorithm that made multiway constraints efficient enough to be solved in real time.

The Garnet way

Garnet currently provides one-way constraints and uses lazy evaluation. Whenever a user changes a variable, all constraints that directly or indirectly depend on that variable are marked as out of date. When a user requests the value of an out-of-date

variable, the constraint solver demands the values of all variables that this constraint depends on. If these variables are out of date, their constraints will in turn be evaluated, and so on. Eventually, the constraint solver reaches variables whose values are either atomic (that is, not computed by a constraint) or up to date, at which point the constraint solver can work its way back to the originally requested variable.

Future versions of Garnet will probably provide a limited form of multiway constraints and eager evaluation. We have found that almost all constraints in interfaces created with Garnet have to be reevaluated when the display is updated; thus lazy evaluation does not avoid the evaluation of enough constraints to justify the increased bookkeeping it requires. Also, the preliminary design for the new multiway algorithm seems to be very efficient.

References

1. A. Borning, "The Programming Language Aspects of Thinglab: A Constraint-Oriented Simulation Laboratory," *ACM Trans. Programming Languages and Systems*, Vol. 3, No. 4, Oct. 1981, pp. 353-387.
2. B.N. Freeman-Benson, J. Maloney, and A. Borning, "An Incremental Constraint Solver," *Comm. ACM*, Vol. 33, No. 1, Jan. 1990, pp. 54-63.
3. A. Borning and R. Duisberg, "Constraint-Based Tools for Building User Interfaces," *ACM Trans. Graphics*, Vol. 5, No. 4, Oct. 1986, pp. 345-374.
4. I.E. Sutherland, "Sketchpad: A Man-Machine Graphical Communication System," *AFIPS Spring Joint Computer Conf.*, 1963, pp. 329-346.
5. P. Barth, "An Object-Oriented Approach to Graphical Interfaces," *ACM Trans. Graphics*, Vol. 5, No. 2, Apr. 1986, pp. 142-172.
6. B.A. Myers, *Creating User Interfaces by Demonstration*, Academic Press, Boston, 1988.
7. T.R. Henry and S.E. Hudson, "Using Active Data in a UIIMS," *Proc. ACM SIGGraph Symp. User Interface Software*, Oct. 1988, pp. 167-178.
8. B. Vander Zanden, "Constraint Grammars — A New Model for Specifying Graphical Applications," *Proc. Conf. Human Factors in Computing Systems (SIGCHI 89)*, Apr. 1989, pp. 325-330.

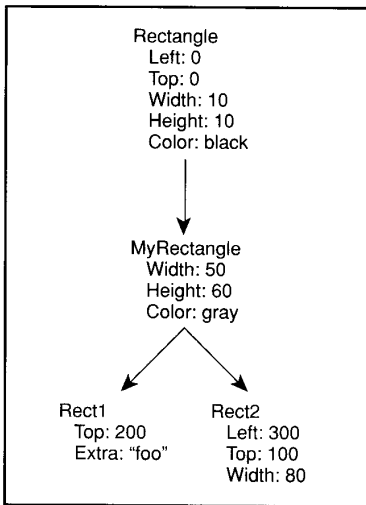


Figure 4. Inheritance in a prototype-instance model. Rect1 and Rect2 are instances of MyRectangle, which is an instance of Rectangle. MyRectangle inherits the left and top of Rectangle. Rect1 inherits all the values of MyRectangle except top, which it overrides. It also adds a new slot. If the left of Rectangle changes, then the lefts of MyRectangle and Rect1 immediately change also. Even though they may be prototypes for other objects, all of these are "real" objects in that they can be displayed on the screen.

Constraint system. A constraint is a relationship among objects that is maintained when any of the objects changes. Constraints are a natural way to express common relationships in graphical user interfaces. For example, in an editor that supports boxes attached by arrows, the designer could specify a constraint that the arrows must stay attached to the boxes. Then, when the boxes are moved by a program or the mouse, the system automatically moves the arrows as well. (The sidebar discusses constraints and their implementation in more detail.)

Coral² was an early version of Garnet's constraints, but it was abandoned because it wasn't fast or flexible enough. Constraints are now integrated with the KR object system.

Constraints in Garnet are arbitrary Common Lisp expressions, stored in object slots. When a program accesses a slot, it cannot tell whether the slot contains a simple val-

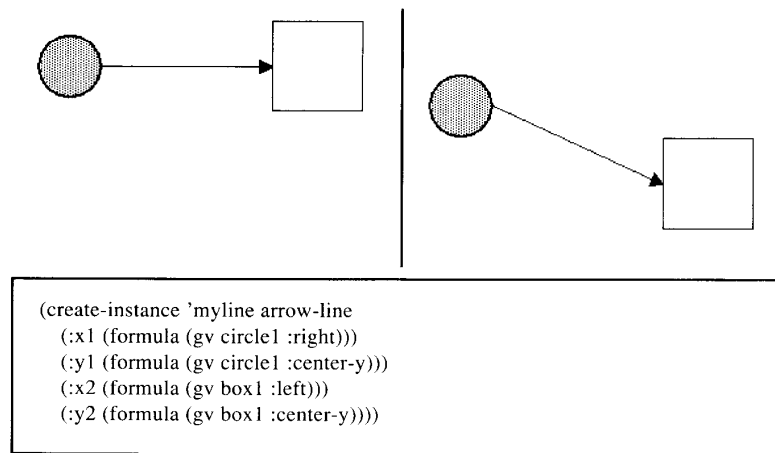


Figure 5. The line stays attached to the box and circle even when they move. Below the graphic is the code to define the constraints on the line.

ue (like a number) or a constraint that calculates the value. Within constraints, references to other objects' slots use the form "(gv other-object slot-name)," where "gv" stands for "get value." Whenever the value of the referenced slot changes, the formula is reevaluated. For example, the code in Figure 5 could be used to keep an arrow attached to two objects.

These formulas are one-way constraints; if the other object changes, the object with the formula is reevaluated, but not vice versa. For example, when the circle or box in Figure 5 moves, the line also moves; however, if the line moves, the circle and box do not move. This type of constraint is also used in Peridot,⁸ Apogee,¹⁰ and many other constraint-based user interface systems. As a special case, Garnet can handle cycles in the constraint dependencies (so object *A* can depend on *B*, and *B* can depend on *A*), in which case Garnet goes around the loop exactly once.

Other systems have used more powerful multiway constraints, which we might add to Garnet later. The primary advantage of the current scheme, however, is that constraint evaluation is very efficient. Garnet can reevaluate more than 3,500 constraints per second on the IBM RT PC implementation, which means that many objects with dozens of constraints can be updated in real time as objects are dragged with the mouse. Of course, Garnet ensures that formulas whose values do not change are not reevaluated, so user interfaces can contain many thousands of constraints.

An interesting and novel feature of Garnet's constraints is that the object referenced in the constraint can be accessed indirectly through a variable. For example, a feedback object in a menu might be constrained to be the same size as whatever object it is on top of. A slot would hold the object that the feedback should appear over. Whenever this slot changes, Garnet automatically reevaluates the formulas that depend on the slot, thus moving the feedback object. To make indirection easy, the gv function can be passed a list of slots to use as indirect references. For example, (gv :SELF :other-obj :left) means: "Look in this object's :other-obj slot. The contents of the slot will be another object. Go to that object and get its left." Since this form is common, (gv-indirect . . .) can be used instead of (gv :SELF . . .).

As an example of indirection, the reverse video rectangle in Figure 6 moves whenever the value of the :obj-over slot changes. This mechanism lets designers keep the input-handling specification independent of the graphics. For example, the interactor object that handles menu behavior (described in the "Input handling" section) simply sets the :obj-over slot to the object that the mouse is over, and the constraints ensure that the graphics that handle feedback change appropriately. The interactor does not need to know anything about the appearance of the graphics.

Because any slot of any object can contain constraints, the use of constraints in Garnet is not limited to the position and

size of graphical objects. Constraints are used throughout the system to control many kinds of behaviors. For example, a constraint can be used to set the mode to determine whether an object will change size or grow based on which mouse button is pressed:

```
(:grow-the-object
 (formula (eq (gv-indirect :which-button)
              :rightdown)))
 ; grow if right button is pressed,
 ; otherwise move
```

Constraints can also be used to connect graphical objects to application-specific objects. For example, the value of a gauge displayed on the screen could be constrained to a temperature data value in the application.

Graphical object system. Opal, Garnet's graphical object system, is designed to make creating and editing graphical objects easy. Opal provides default values for all object properties, so simple objects can be drawn by specifying only the necessary parameters. For example, to create a rectangle at (10,20) with size 30 by 40, a designer need only write:

```
(create-instance 'myrect rectangle
 (:left 10) (:top 20) (:width 30)
 (:height 40))
```

The object system is integrated with the constraint system, so that any object property can be specified with formulas instead of numbers, as shown in the code in Figure 5.

Opal also automatically handles object drawing and erasing. If any object property changes, Opal automatically refreshes the screen and redraws that object. If that object overlaps others on the screen, Opal ensures they are also redrawn correctly (see Figure 7). In addition, if the modifications change other objects due to constraints, these other objects are also redrawn automatically.

Rather than simply redrawing all the objects, Opal tries to minimize the number of objects that are erased and redrawn. This is very important for complex scenes. For example, moving one object through a window containing 200 other objects takes 25.6 milliseconds per move (39 moves per second) rather than the 568 milliseconds (1.76 moves per second) it would take if Opal redraw all the objects.

Because Opal knows where all objects are, it can also handle window refresh (when the window is uncovered). Opal's

Computer

Byte

IEEE Software

CACM

IEEE Spectrum

Computer

Byte

IEEE Software

CACM

IEEE Spectrum

```
(create-instance 'feedback-box Rectangle
 (:obj-over NIL)
 ; The object that should be
 ; highlighted
 (:visible (formula (gv-indirect :obj-over))) ; I am visible if there is an object
 ; in :obj-over
 (:left (formula (gv-indirect :obj-over :left))) ; The size and position is the same
 ; as whatever I am over
 (:top (formula (gv-indirect :obj-over :top)))
 (:width (formula (gv-indirect :obj-over :width)))
 (:height (formula (gv-indirect :obj-over :height)))
 (:draw-function :xor) ; XOR this rectangle
```

Figure 6. Setting the :obj-over slot of the feedback-box rectangle makes it appear over that object because of the constraints on the size and position.

clients never have to worry about the window system's refresh events. In fact, Opal and the interactors completely hide all window system functions.

To make an object appear in Opal, the designer only has to add it to an Opal window. Removing the object from the window makes the object disappear. To change an object's color or size, the designer sets the appropriate slots. Therefore, the programmer never calls the draw or erase methods on objects directly; these methods are only called from internal Opal routines. In this respect, Opal departs significantly from other graphical object systems.

An important aspect of Opal is its ability to group graphical objects into collections called *aggregates*. When an aggregate is used as a prototype, its instances contain copies of the entire collection of objects. Thus, there is an instance for each component of the aggregate as well as for the aggregate itself (see Figure 8). Changes to the aggregate are immediately reflected in

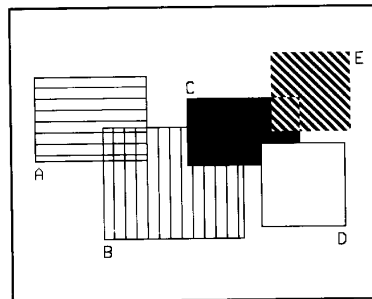


Figure 7. Opal lets graphics overlap opaquely and automatically refreshes damaged parts. If C is erased, B, D, and E will need to be redrawn. Opal uses a clipping region so that only the parts of B, E, and D that overlap C are affected, and so that A does not have to be redrawn. B is drawn with the OR drawing function, E is drawn with XOR, and the others are drawn with Copy.

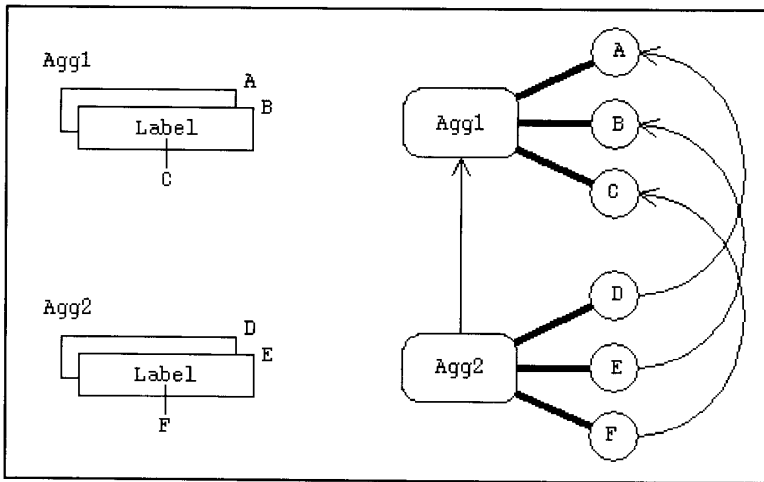


Figure 8. A, B, and C are components of the aggregate Agg1. Making an instance of Agg1 automatically makes instances of each of its components. The dark lines are component links, and the arrow lines are instance links. If Agg1 or any of its components change, Agg2 and its components would change automatically.

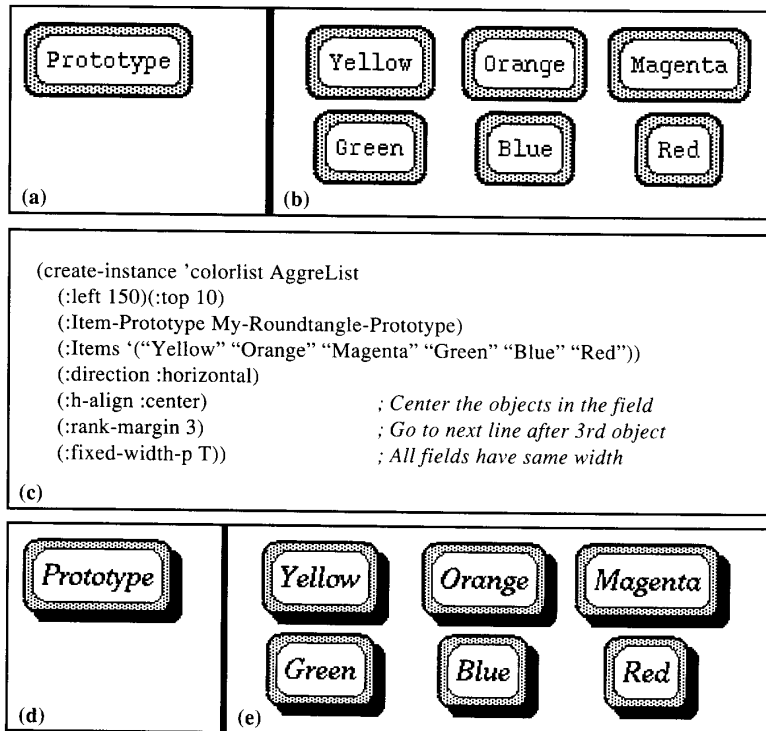


Figure 9. A prototype for the menu items (a) and a two-dimensional aggregate using instances of that prototype for each element (b). The actual code to display the menu is shown in (c). If the prototype is subsequently edited to add a drop shadow and change the font (d), all instances immediately inherit the new values and structure (e).

all instances, including when components are added or deleted from the prototype, in which case the corresponding components are immediately added or deleted from all instances. Previous implementations of the prototype-instance model have not supported such structural changes to instances.

As an example, consider a graphical tool that lets the user design menus. The user might draw a prototype button (Figure 9a) and then create many instances of that button in the interface (Figure 9b). If the prototype is subsequently edited—perhaps to add a drop shadow and change the text font (Figure 9d)—Opal insures that all instances immediately inherit the changes (Figure 9e). This requires that Opal add a new object for the shadow to each instance.

Of course, each instance can override any of the slots, making it easier to build prototypes for complex, composite objects like menus and scroll bars. The prototype contains example values for slots (like the text strings in the menu), and the instances override these values.

A special form of aggregate is an *aggrelist*, which is used to lay out a list or table of elements. An aggrelist contains a single prototype for all the elements, and each element uses a different value for some value of that prototype. For example, menus are usually implemented as an aggrelist with a single prototype for the items but with a different string value displayed in each item. The prototype in Figure 9a is an aggregate containing the two rounded rectangles (one grey and one white) and the example string "Prototype." Aggrelists allow the items to be displayed horizontally, vertically, and in multiple rows, or the programmer can specify a layout (as for the gauge labels shown later in Figure 12f). We are also working on the implementation of aggregategraphs and aggregretrees to lay out graphs and trees.

Input handling. One of the most difficult tasks when creating highly interactive user interfaces is handling the mouse, keyboard, and other input devices. Window managers and user interface toolkits typically provide only a stream of device-dependent mouse positions and keyboard events, and they require that the programmers handle all interactions themselves. Garnet provides significantly more help through the use of interactors, which are encapsulations of input device behaviors.³

There are few distinct behaviors in user interfaces. For example, although graphics can vary significantly and the specific mouse buttons can change, all menus operate in

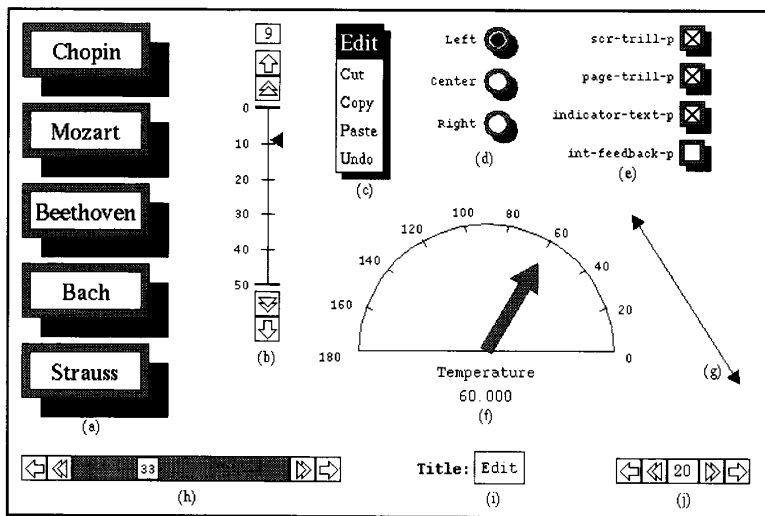


Figure 12. Some of Garnet's gadgets: (a) floating buttons, (b) a number slider, (c) a menu, (d) floating radio buttons, (e) floating check boxes, (f) a semicircular gauge, (g) an arrow-line, (h) a scroll bar, (i) a labeled text entry field, and (j) a number entry field.

```
(create-instance 'color-selector
  menu-integer
  (:start-where '(element-of colorlist))
  ; Colorlist is defined in Figure 9c
  (:feedback-obj feedback-box))
; Feedback-box is defined in Figure
; 6 and will appear over whichever
; of the items the mouse is over
```

The feedback can be a set of objects, for example, the selection handles in Figure 1. As shown in Figure 11, the feedback does

not even have to be a separate object. Instead, some property of the object itself can change in response to the mouse. If the `:feedback-obj` is not supplied, the menu interactor sets the `:interim-selected` slot of the object the mouse is over and the `:selected` slot of the item the mouse ends up on. The menu in Figure 11c has constraints that change the position of the object under the mouse, based on whether the value of `:interim-selected` is T or NIL (true or false). The menu in Figure 11e decides whether to

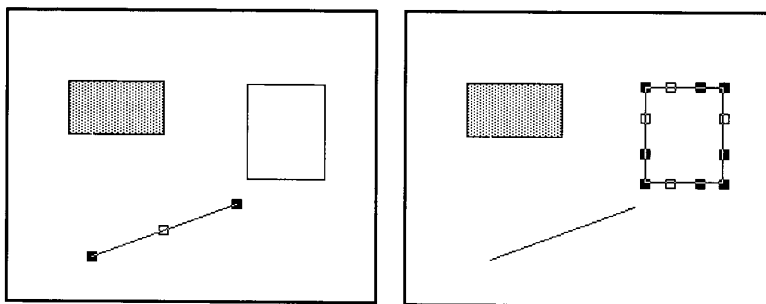


Figure 13. The graphics selection gadget shows control handles around the selected object. Pressing on a white handle moves the object, and pressing on a black one changes the object's size. When a line is selected, only three control points are shown: the black ones change the end points and the white one moves the line, keeping the same length and slope.

use a Roman, bold, italic, or bold-italic font based on the values of both `:interim-selected` and `:selected`.

An important feature of interactors is that a single interactor can handle a set of objects. For example, there is a single interactor for each menu, rather than one for each menu item. Similarly, a single move-grow interactor can be used for an entire set of objects that can be moved with the mouse (as in Figure 1). The interactor will choose which object to move based on where the mouse button is pressed.

The interactors are based on Smalltalk's Model-View-Controller.¹² In Garnet, the application objects correspond to the model, the Opal graphical objects correspond to the view, and the interactors correspond to the controller. In Smalltalk, however, if a programmer wants a new style of object, he or she usually has to write code for all three parts. In Garnet, programmers almost never need to create new forms of interactors, even for application-specific objects; it is sufficient to create an instance of one of the supplied types of interactors and then supply appropriate parameters to achieve the desired behavior. New types of interactors would be needed in Garnet only for radically different types of behaviors, such as gesture recognition or new kinds of physical input devices (such as a 3D joystick). Constraints in Garnet connect the models, views, and controllers.

Gadgets. On top of Opal and the interactors is a collection of interaction techniques — called gadgets — that provide a starting point for applications. There are gadgets for menus, scroll bars, buttons, gauges, etc. Figure 12 shows some of the gadgets supplied with Garnet (more are being created). These can be used by designers who do not want their applications to have a custom look and feel. Most of the gadgets have a number of parameters to let designers vary many aspects of the appearance and behavior.

In some toolkits, such as Xtk and Interviews, each gadget is a window. This significantly limits what gadgets can do. In Garnet, however, gadgets are not windows, so there is a gadget, for example, that is a line with an attached arrowhead (Figure 12g). An even more sophisticated gadget in Garnet handles object selection (see Figure 13). It displays selection handles around the selected object and lets the user move or grow the object by pressing on the handles. Therefore, to support graphical object selection, a designer using Garnet need only create an instance of an object-selec-

essentially the same manner. Another example is the way objects move when being dragged with the mouse. The interactors capture these common behaviors in a central place while still allowing a high degree of customizing by application programs.

Other advantages of the interactors are:

- They are entirely look independent; any graphics can be attached to a particular feel (see Figures 10 and 11).
- They let designers separate the details of object behavior from the application and the graphics (long a goal of user interface software design).
- They support multiple input devices operating in parallel.
- They let users operate on any of a number of different applications running in separate windows in the same Lisp process and same address space, thereby supporting a form of multiprocessing.
- Opal and the interactors hide the complexities of X Windows graphics and event handling, making Garnet easier to use than X Windows and allowing Garnet to be ported to other graphics packages, such as Macintosh Quick Draw or Display Postscript, without changing Garnet applications.

There are only six types of interactors in Garnet, and these cover all the kinds of interactions used in graphical user interfaces:

- *Menu-Interactor*. For choosing one or more items from a set, or for a single stand-alone button.
- *Move-Grow-Interactor*. For moving or changing the size of an object or one of a set of objects using the mouse. This interactor can be used for one- or two-dimensional scroll bars and horizontal and vertical gauges, and for moving or growing application objects in a graphics editor.
- *New-Point-Interactor*. For entering one, two, or an arbitrary number of new points using the mouse; for example, for creating new lines or rectangles in an editor.
- *Angle-Interactor*. For calculating the angle at which the mouse moves around some point. It can be used for circular gauges or for rotating objects.
- *Trace-Interactor*. For capturing all the points the mouse goes through between start and end events, as needed for free-hand drawing.
- *Text-String-Interactor*. For inputting a small (optionally multiline) string of text.

Each interactor is parameterized in various ways, so the programmer can control the mouse or keyboard events that make it

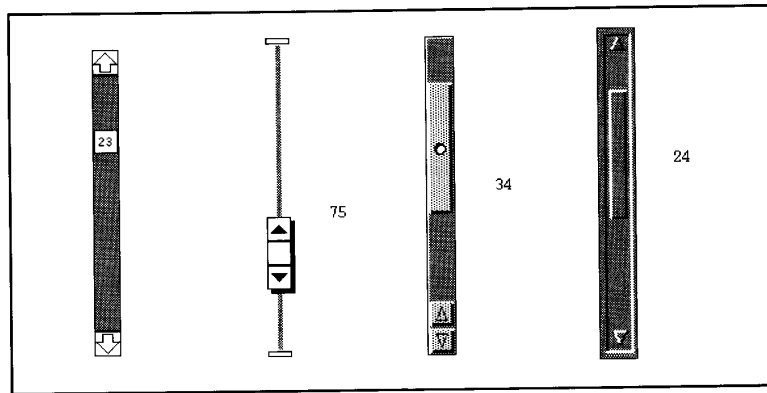


Figure 10. The same type of interactor can handle different graphic looks. Here, move-grow-interactors for the indicators and menu-interactors for the arrow buttons handle scroll bars that look like those in the Macintosh, Open Look, Next, and OSF/Motif environments.

start and stop and the optional application procedures to be called on completion. The most significant parameters, however, are the objects where the interactor operates and the (optional) objects to handle feedback. Each type of interactor controls the

graphics with a well-defined protocol. For example, the menu interactor sets the :obj-over slot of the feedback object. Therefore, to turn the list of items in Figure 9b into a menu using the feedback object from Figure 6, we need only the following code:

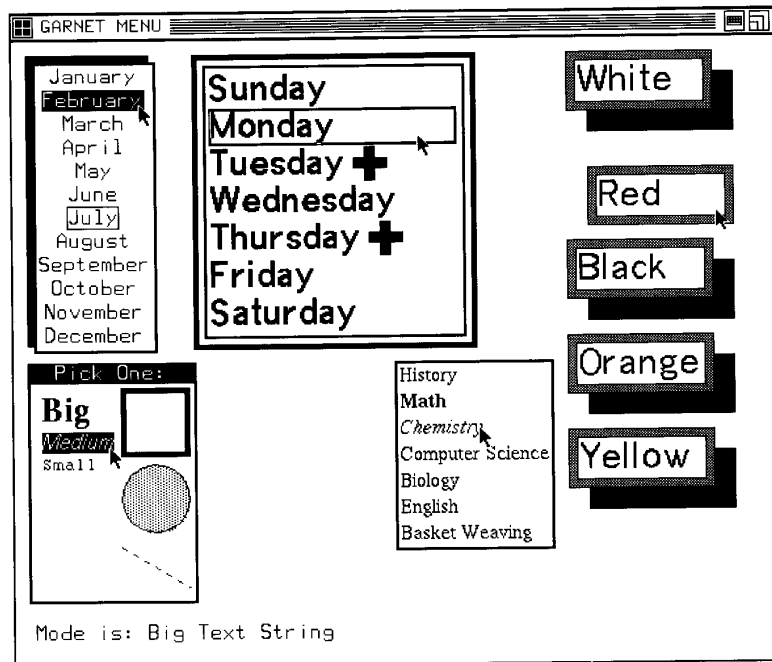


Figure 11. The menu-interactor can handle menus with many different looks and feels. The items in (c) simulate floating by moving when the mouse is over them. The items in (e) change to italic when the mouse is over them; the final selection is bold.

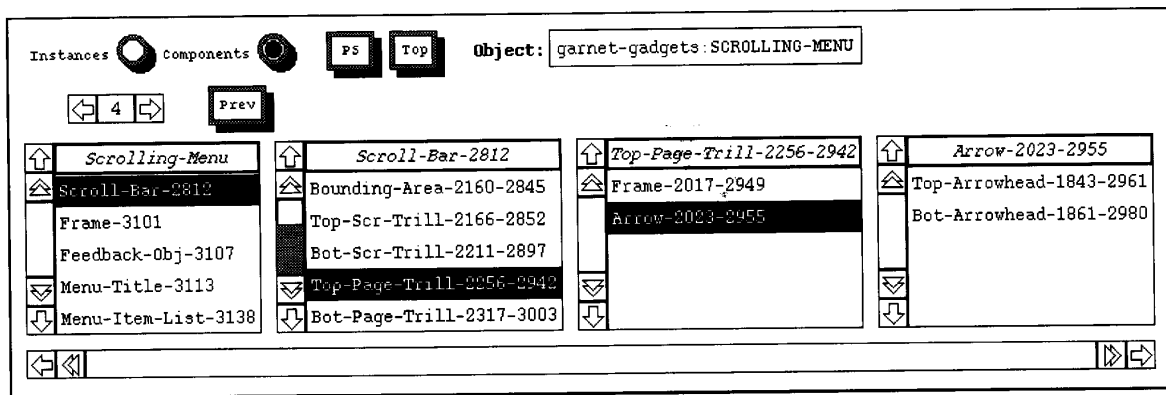


Figure 14. A browser that will show either the components of an aggregate or the instances of a prototype. We created this with the Garnet Toolkit, and it can be used to debug Garnet code.

tion gadget (assuming he or she finds the Garnet look and feel acceptable).

Since creating new gadgets with Garnet is easy, we expect many designers to create their own gadgets rather than use the supplied ones. By using Lapidary (detailed in the next section) or by programming using Opal and interactors, programmers can create new styles of menus and buttons in minutes.

Debugging tools. Defining user interfaces with constraints and interactors is natural and effective, but it can be difficult to find bugs in the code. Since with constraints, a bad value in one place can cause many different objects to have bad values, the effects of a bug are not local.

Garnet provides a large and growing number of debugging tools to help with program implementation. These tools provide tracing mechanisms to show how a slot got its current value, where objects are on the screen or why they are not visible, and what happens when the mouse or keyboard is used. The tools also provide extensive checking for legal values of graphics slots and convenient ways to inspect objects, including browsers (see Figure 14).

The interface builder

Lapidary. On top of the Garnet Toolkit layer are a number of tools to make creating user interfaces easier. The most important is the Lapidary interface builder.⁴ Lapidary provides a graphical front end to most of the underlying toolkit features, so

that a program's graphical aspects can be specified pictorially. In addition, the behavior of these objects at runtime can be specified using dialogue boxes and by demonstration.

In particular, Lapidary lets the designer, who does not have to be a programmer, draw pictures of application-specific graphical objects that the application will create and maintain at runtime. These objects include the graphical entities the end user will manipulate (such as the components of the picture), the feedback showing which objects are selected (such as small squares that serve as handles on the sides and corners of an object), and the dynamic feedback objects (such as hairline boxes to show where an object is being dragged). The designer creates prototypes of the objects in Lapidary, and the application program then creates instances of the prototypes as needed.

In addition, Lapidary supports the construction and use of gadgets, such as menus, scroll bars, buttons and icons. Lapidary therefore supports both the use of a pre-defined library of gadgets and the definition of new gadgets with a unique look and feel. Both types can even be combined in the same application. The designer can specify the runtime behavior of these objects in a straightforward way using constraints and abstract descriptions of their interactive response to the input devices. Lapidary generalizes from the specific example pictures to let the designer specify the graphics and behaviors by demonstration.

The designer can specify the behavior of objects in various ways. Graphical con-

straints can be attached to objects using iconic menus (see Figure 15). If an object should move with the mouse, it can be selected and declared a feedback object. Lapidary will automatically generalize the constraints on the feedback object so that they refer to whatever graphical object the mouse is over. For example, in Figure 15b the check mark is constrained to the Stop button, but Lapidary generalizes this constraint to use a variable (like the :obj-over slot of Figure 6) so that the check mark can appear next to any of the buttons.

Also, if an object should change based on some user action, the designer can specify this by demonstration. The designer draws one state, then another, and Lapidary automatically constructs the constraints to change the object between the two states. The menus of Figures 11c and 11e can be defined in this way.

Consider how a designer might create the boxes and arrows for a graph editor. First, the designer draws a picture of the boxes, say, using rounded rectangles (Figure 16a). Then, he or she creates an example text label and uses the iconic constraint menus to center it at the top (Figure 16b). This will serve as the prototype for the boxes the application will create, so the designer selects the entire group and saves it as a named prototype. Lapidary asks the designer for the object parameters, so the designer specifies the position and size of the box and the string for the label. Lapidary then writes this prototype to a file and defines it in memory (in case the application is running concurrently with Lapidary).

Next, the designer creates a copy of the

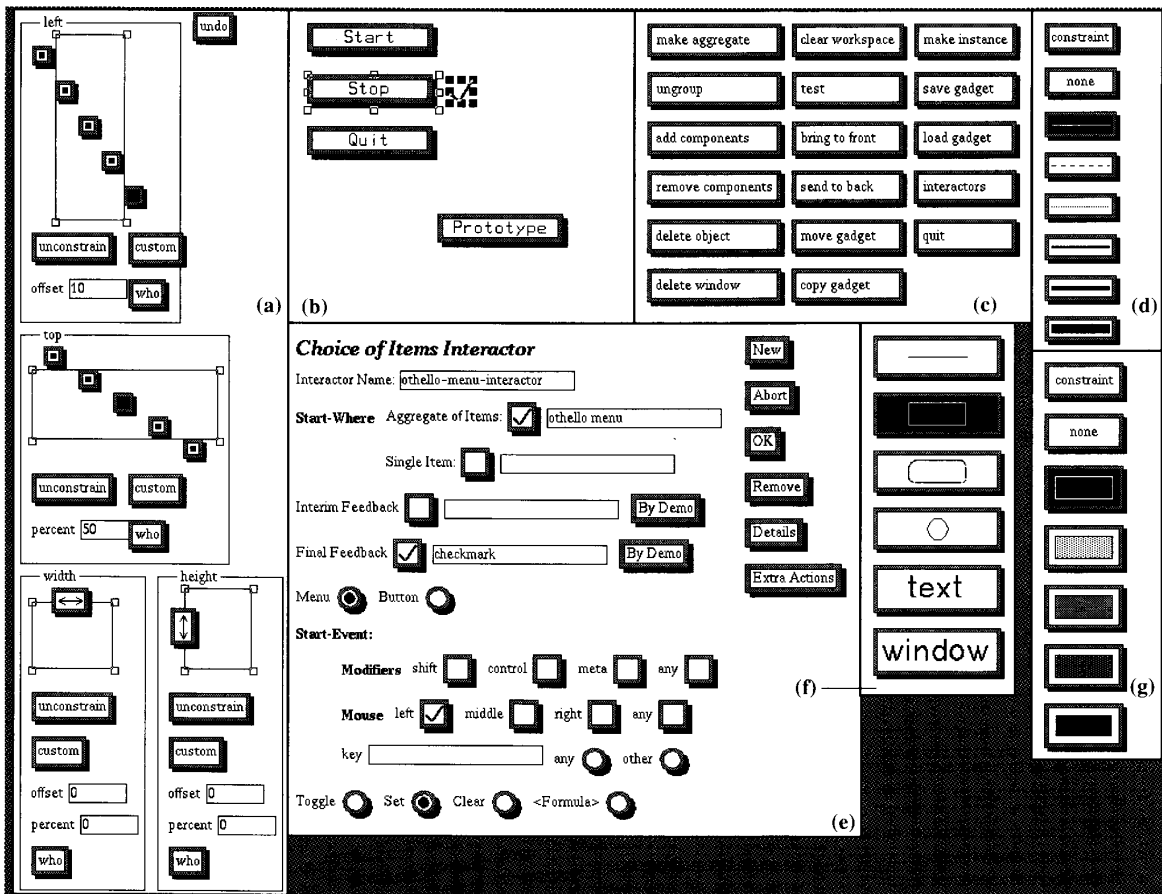


Figure 15. Lapidary in action. The work window (b) contains a prototype and three buttons made from it. The check mark icon is the primary selection (indicated by black squares) and the center button is the secondary selection (indicated by white squares). The iconic constraint menu (a) shows that the primary selection is constrained to be 10 pixels to the right of the secondary selection and centered by it vertically. Window (c) contains the main Lapidary commands. The menus on the right determine the type of the next object to be created (f), the line style (d), and the fill style (g). The large window in the center (e) is a dialogue box for specifying the behavior of the menu being created; it shows that the check mark icon will be the final feedback to show which item is selected when the left mouse button is pressed. This dialogue box was created using Jade.

box, draws an arrow-line (Figure 16c), and uses the line constraint menu to specify that the ends of the line should be centered in the boxes (Figure 16d). Then the designer selects only the line and saves it as a prototype. Lapidary notes that the prototype has constraints to objects that are not being saved, so it asks the designer if the objects are to be parameters of the line prototype. The designer specifies that the parameters should be called from-obj and to-obj. Now the application can create instances of the lines, supplying the objects for the from-obj and to-obj slots, and new lines will appear in the application win-

dow. The application knows nothing about the graphics used for the lines, and the designer did not have to write any graphics code. This entire design session takes about four minutes.

Jade. It is sometimes easier to list the contents of a dialogue box or menu than to meticulously draw it. The Jade dialogue box creation system automatically creates a dialogue box or menu from a specification of its contents.⁵ In addition to being simple to work with, the specification passed to Jade has the additional advantage of being independent of any particular look

and feel. The textual specification has a Lisp-like syntax and lists only the string labels to appear in the dialogue box, the type of input required (such as the choice of one item of a set, or of a number in a range). When Jade is given a particular look and feel (Macintosh, Motif, Garnet standard, etc.), it can choose the correct interaction techniques, which themselves can be designed using Lapidary. The heuristic rules that determine the placement of various parts of the interface are specific to a particular look and feel. For example, the set of buttons that make a dialogue box go away ("OK," "Cancel," etc.) will be at the

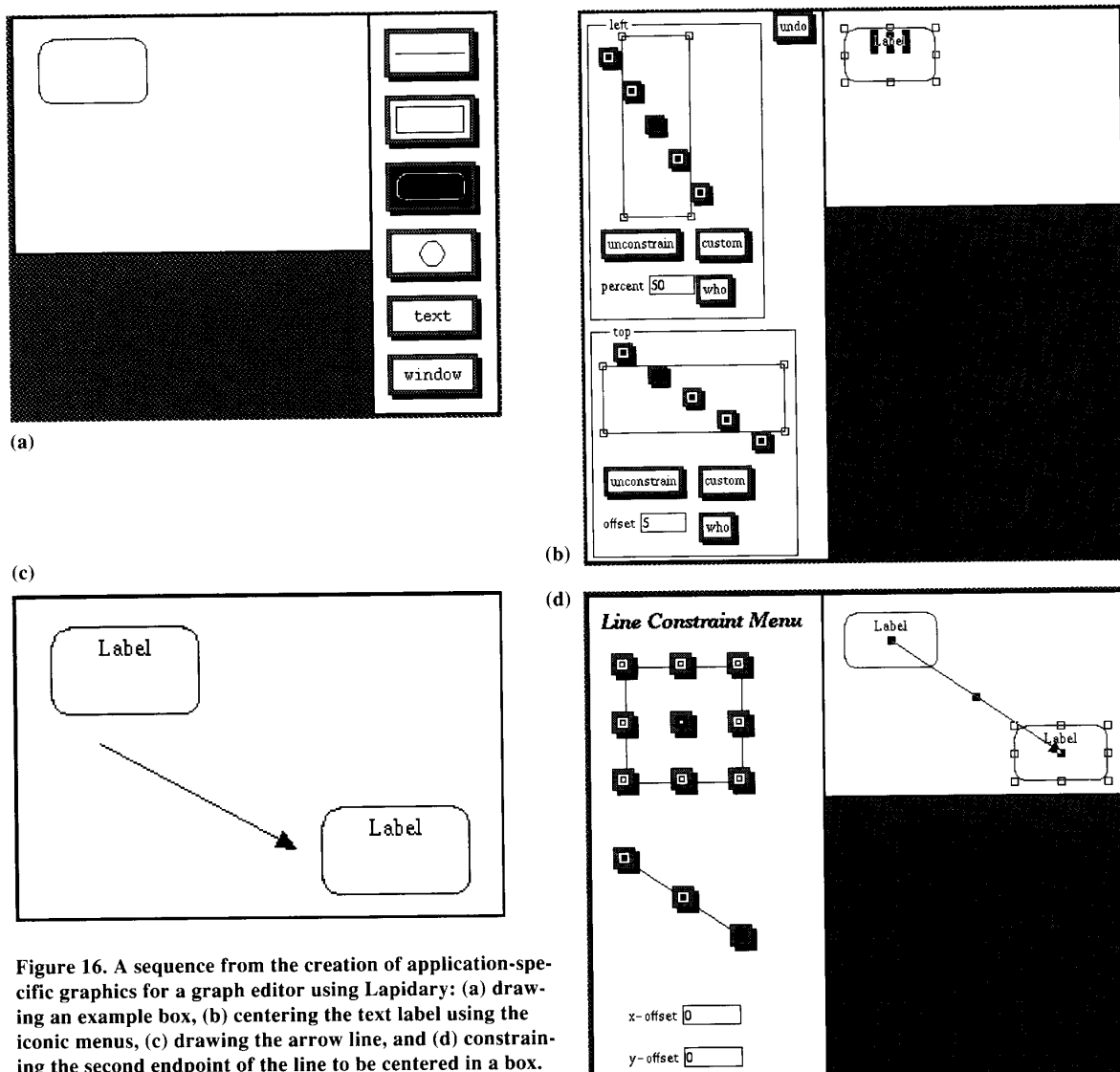


Figure 16. A sequence from the creation of application-specific graphics for a graph editor using Lapidary: (a) drawing an example box, (b) centering the text label using the iconic menus, (c) drawing the arrow line, and (d) constraining the second endpoint of the line to be centered in a box.

right for a Macintosh-like dialogue box and at the top for a Xerox Star-like one. Jade created the dialogue box in Figure 15e automatically.

The heuristics for placing objects in dialogue boxes attempt to create an attractive display based on graphic arts principles. If the designer is not happy with the dialogue box, however, he or she can edit it in Lapidary, adding decorations (such as extra rectangles and lines) and moving parts around. These changes are saved in an exceptions file so they can be reapplied even if the original textual specification is edited and the dialogue box regenerated.

C32 spreadsheet. Although many desired constraints can be specified using Lapidary's iconic menus, designers occasionally need more powerful constraints. The C32 spreadsheet program in Garnet lets designers enter arbitrary Lisp constraint expressions. It provides many of the advantages for graphics that financial spreadsheets provide for business. In particular, C32 lets the designer monitor and debug interfaces by watching spreadsheet values while the user interface is running. It also provides extensible command menus, automatic generation of appropriate object references from mouse clicks in graphics

windows, and automatic generalizations of example constraints so they can be used in multiple formulas. Figure 17 shows a typical C32 session.

The Garnet Toolkit is operational and has many local and external users. More than 80 companies and universities are licensed to use Garnet. (Garnet is available free from Carnegie Mellon University. Designers interested in using Garnet should contact Myers at the CMU address listed after the biographies.) We are working to increase the toolkit's

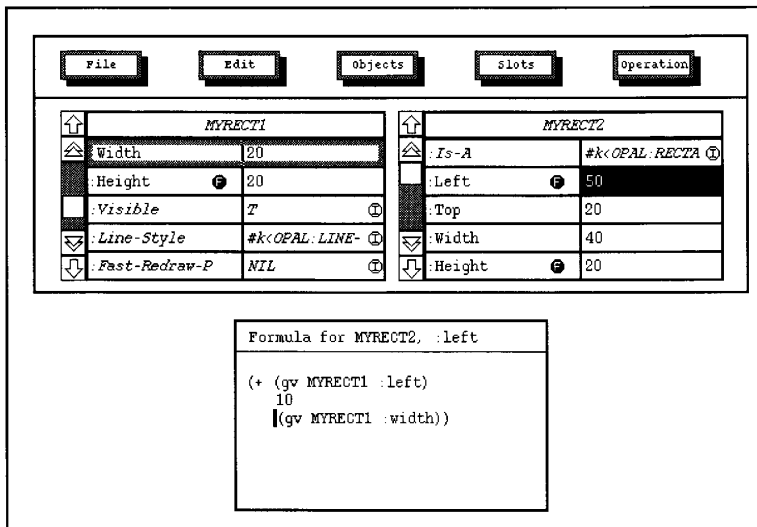


Figure 17. Viewing two objects in the Garnet C32 spreadsheet. At the top of the window is a set of pull-down menus. Each object has its own set of columns. The slot names of the objects are at the left of each column, along with the current values. If the value is computed by a formula, an icon with an "F" is shown. Inherited values are in italics and marked with a different icon. At the bottom is a formula display window for the selected slot. When the user clicks on a slot or an object in a graphics window, a reference to that object is inserted into the formula.

functionality and performance and to find new ways to support the design and implementation of application user interfaces. We hope the toolkit will be widely distributed and used in the Common Lisp community. Lapidary is working but is not yet releasable. Jade is partially working and has been used to generate the Lapidary dialogue boxes. The C32 system is just now being implemented.

We plan to work on allowing more of the interface to be specified by demonstration rather than through dialogue boxes and coding. For example, Lapidary could infer graphical constraints and mouse dependencies from the user's drawing, in a manner similar to Peridot.⁸ We also hope to extend Garnet to handle other input and output technologies, such as physical dials and switches, speech input and output, and gesture recognition.

Finally, we plan to provide a high-level graphical editor framework that provides many of the operations common in systems that let the user create and manipulate graphical objects. These might include support for a palette of object types; selecting objects; changing object size, position, and other properties; deleting objects; au-

tomatically laying out objects as a list, table, graph, or tree; saving and restoring objects to a file; printing; and undoing. For many programs, this subsystem will provide most of the standard functionality, and designers will need to specify only the application-specific parts.

Garnet has only been working for a short time, but it has already demonstrated that it makes the creation of graphical, highly interactive user interfaces easier. For example, in an informal experiment, creating a simple graphical editor like that in Figure 1 took various people two to four hours with the Garnet Toolkit but 10-20 hours with other toolkits such as Apple's MacApp and Sun's Open Look. When Lapidary and the other high-level tools are released, we expect programmer productivity to increase even more. ■

Acknowledgments

In addition to the authors, others who have helped design and implement Garnet include Pedro Szekely, Pavan Reddy, Jake Kolojejchick, and Lynn Baumeister. Amy Moormann Zarem-

ski and Pedro Szekely were brave early users of the Garnet Toolkit and helped us debug it. Bernita Myers, Amy Moormann Zaremski and the referees provided helpful comments on this article.

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

References

1. B.A. Myers, "User Interface Tools: Introduction and Survey," *IEEE Software*, Vol. 6, No. 1, Jan. 1989, pp. 15-23.
2. P.A. Szekely and B.A. Myers, "A User Interface Toolkit Based on Graphical Objects and Constraints," *Proc. ACM Conf. Object-Oriented Programming, Systems Languages, and Applications in SIGPlan Notices*, Vol. 23, No. 11, Nov. 1988, pp. 36-45.
3. B.A. Myers, "Encapsulating Interactive Behaviors," *Proc. Conf. Human Factors in Computing Systems (SIGCHI 89)*, Apr. 1989, pp. 319-324.
4. B.A. Myers, B. Vander Zanden, and R.B. Dannenberg, "Creating Graphical Objects by Demonstration," *Proc. ACM SIGGraph Symp. User Interface Software and Technology*, Nov. 1989, pp. 95-104.
5. B. Vander Zanden and B.A. Myers, "Automatic, Look-and-Feel-Independent Dialogue Creation for Graphical User Interfaces," *Proc. Conf. Human Factors in Computing Systems (SIGCHI 90)*, Apr. 1990, pp. 27-34.
6. B.A. Myers et al., "The Garnet Toolkit Reference Manuals: Support for Highly Interactive, Graphical User Interfaces in Lisp," Tech. Report CMU-CS-90-117, Carnegie Mellon University, Computer Science Department, Mar. 1990.
7. M.A. Linton, J.M. Vliissides, and P.R. Calder, "Composing User Interfaces with Interviews," *Computer*, Vol. 22, No. 2, Feb. 1989, pp. 8-22.
8. B.A. Myers, *Creating User Interfaces by Demonstration*, Academic Press, Boston, 1988.
9. A. Borning, "Thinglab --- A Constraint-Oriented Simulation Laboratory," Tech. Report SSL-79-3, Xerox Palo Alto Research Center, July 1979.
10. T.R. Henry and S.E. Hudson, "Using Active Data in a UIMS," *Proc. ACM SIGGraph Symp. User Interface Software*, Oct. 1988, pp. 167-178.

11. C. Wiecha et al., "Generating User Interfaces to Highly Interactive Applications," *Proc. Conf. Human Factors in Computing Systems (SIGCHI 89)*, Apr. 1989, pp. 277-282.
12. G.E. Krasner and S.T. Pope, "A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System," *J. Object-Oriented Programming*, Vol. 1, No. 3, Aug. 1988, pp. 26-49.



Roger B. Dannenberg is a research computer scientist at Carnegie Mellon University. His research interests include programming language design and implementation and the application of computer science techniques to the generation, control, and composition of computer music.

Dannenberg received a BA from Rice University in 1977, an MS from Case Western Reserve University in 1979, and a PhD from Carnegie Mellon in 1982. He is a member of Phi Beta Kappa, Sigma Xi, Tau Beta Pi, Phi Mu Alpha, the ACM, SIGCHI, and the Computer Music Association.



Edward Pervin is a research programmer at Carnegie Mellon University, where he is working on Garnet. Pervin received his MS and BSc degrees in mathematics from Carnegie Mellon.



Brad A. Myers is a research computer scientist at Carnegie Mellon University, where he is the principal investigator for Garnet. His research interests include user interface development systems, user interfaces, programming by example, visual programming, interaction techniques, window management, programming environments, debugging, and graphics.

Myers received the MS and BSc degrees from the Massachusetts Institute of Technology, and he received a PhD in computer science at the University of Toronto, where he developed the Peridot user interface management system. He is a member of the IEEE, the Computer Society, the ACM, SIGGraph, and SIGCHI.

Brad Myers can be reached at the School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3890. Electronic mail can be sent to garnet@cs.cmu.edu.



Brad Vander Zanden is an assistant professor at the University of Tennessee and an active participant in the Garnet project. He was a post-doctoral fellow at Carnegie Mellon University for two years. His interests include user interface development systems, user interfaces, constraint systems, incremental constraint satisfaction, programming environments, and graphics.

Vander Zanden received his BS degree from Ohio State University, and his MS and PhD degrees from Cornell University. He is a member of the IEEE, the Computer Society, the ACM, SIGGraph, and SIGPlan.



Andrew Mickish received a BS degree from Carnegie Mellon University, where he worked on the Garnet project as an undergraduate. He plans to enter graduate school to pursue interests in user interfaces and artificial intelligence.



Dario A. Giuse is a senior computer scientist at Carnegie Mellon University. His research interests include user interface development environments, constraint satisfaction, graphics, knowledge representation systems, tutoring systems for foreign languages, medical informatics, and interactive knowledge acquisition systems.

Giuse received the Dr.Ing. degree (summa cum laude) in computer science from the Politecnico di Milano, Italy, in 1979. He is a member of the IEEE Computer Society, the ACM, the AAAI, and the Mathematical Association of America.



David S. Kosbie is a graduate student at Carnegie Mellon University, where he is a member of the Garnet project. His research interests include user interface development systems, user interfaces, and methods for simplifying graphical programming. Kosbie received the BA degree from Harvard University in 1987. He is a member of the IEEE and the Computer Society.



Philippe Marchal is a member of the Human-Computer Interaction group of the European Computer-Industry Research Center in Munich. He is a former visiting scientist at Carnegie Mellon University, where he was a researcher in the Garnet project. His research interests include user interfaces, user interface management systems, interaction techniques, human factors, graphics, and object-oriented programming. Marchal graduated from the University of Grenoble, France.