Comprehensive Support for Graphical, Highly-Interactive User Interfaces: The Garnet User Interface Development Environment

Brad A. Myers,
Dario Giuse, Roger B. Dannenberg, Brad Vander Zanden, David Kosbie,
Ed Pervin, Andrew Mickish, Philippe Marchal

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

May 1, 1990

Abstract

The Garnet User Interface Development Environment contains a comprehensive set of tools that make it significantly easier to design and implement highly-interactive, graphical, direct manipulation user interfaces. The lower layers of Garnet provide an object-oriented, constraint-based graphical toolkit that allows properties of graphical objects to be specified in a simple, declarative manner and then maintained automatically by the system. The dynamic, interactive behavior of the objects can be specified separately by attaching high-level ‘‘interactor’’ objects to the graphics. The higher layers of Garnet include an interface builder tool, called Lapidary, that allows the user interface designer to draw pictures of *all* graphical aspects of the user interface. Unlike other interface builders, Lapidary allows toolkit items, such as menus and scroll bars, to be created as well as used, and Lapidary also allows application-specific graphical objects (the *contents* of the application’s window) to be created in a graphical manner. Other high level tools include an automatic dialog box and menu editor, and a spreadsheet program for specifying complex graphical constraints. This paper provides an overview of the entire Garnet system.

This article was printed as ‘‘Garnet; Comprehensive Support for Graphical, Highly-Interactive User Interfaces,’’ *IEEE Computer*. Vol. 23, No. 11. November, 1990. pp. 71-85. Translated into Japanese and reprinted in *Nikkei Electronics*, No. 522, March 18, 1991, pp. 187-205.

1. Introduction

The Garnet research project is creating a set of tools that make it easy to design and implement highly-interactive, graphical, direct manipulation user interfaces. In addition, Garnet is ideal for rapidly prototyping different interfaces, and for exploring various user interface metaphors during early product design.

Garnet has a number of important features that differentiate it from other user interface tools, including an emphasis on handling the run-time *behavior* of objects (how they change when the user operates on them), and on handling *all* visual aspects of the user interface for programs, including the graphics displayed by the program and the contents of all application-specific windows.

For example, when creating an application that allows the user to manipulate boxes connected by arrows (such as a graph editor or a project-planning chart—see Figure 1), Garnet's user interface builder, called Lapidary, allows the designer to draw pictures of what the boxes and arrows should look like and demonstrate how they should respond to the mouse. Menus, palettes and dialog boxes can be generated automatically, or added graphically using Lapidary. Toolkits and interface builders in other systems might help build the menus and palettes, but not with implementing and managing the boxes and arrows themselves. Therefore, these must be coded directly using the underlying graphics package and input device handlers. Typically, coding the application-specific graphics is 10 to 100 times more effort than dealing with the menus and buttons, so Garnet will often save programmers significant time.

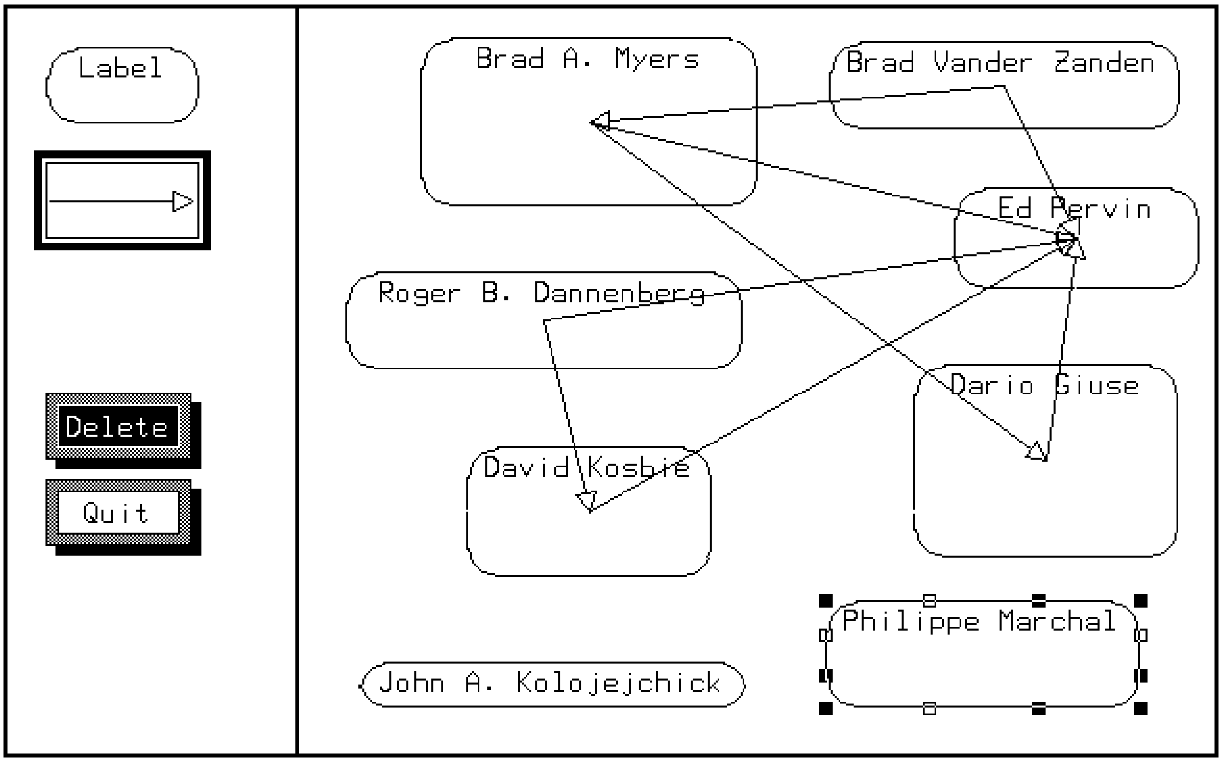


Figure 1: A simple boxes-and-arrows editor created using Garnet. An arbitrary number of new boxes and arrows can be added, and their initial position is specified using the mouse. Any existing box or arrow can also be selected, changed position or size, and deleted. This entire editor was implemented in about 3 hours using the Garnet Toolkit.

Garnet, which stands for Generating an Amalgam of Real-time, Novel Editors and Toolkits, is entirely ‘‘look-and-feel independent,’’ which means that the user interface designer can create user interfaces with an original graphical appearance and behavior, or choose from a set of predefined appearances and behaviors.

Garnet contains both low-level and high-level tools. The low level tools are called the ‘‘Garnet Toolkit,’’ and use a number of innovative mechanisms to make it easier to implement user interfaces. These include 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 handles refreshing the display when objects change), and separate specification of input handling from graphics.

The high-level Garnet tools include the Lapidary interface builder, which lets the user interface designer draw pictures of what the user interface should look like and then demonstrate how it should change in response to user input; the Jade Dialog Box creation system, which automatically creates menus and dialog boxes from a list of their contents; and the C32 spreadsheet for specifying complex constraints. This paper provides an overview of all parts of Garnet. Previous papers[1, 2, 3, 4] have concentrated on individual aspects of the system. There is also a complete reference manual for the Garnet Toolkit[5].

Garnet is implemented in Common Lisp and uses the X window manager, through the standard CLX interface from Common Lisp to X. Garnet is therefore portable and runs on various machines and operating systems. So far, Garnet runs on CMU, Lucid, Allegro and TI Common Lisps, and on various versions of the X/11 window manager. Garnet does *not* use the Common Lisp Object System (CLOS) or any Lisp or X toolkit (such as InterViews[6], CLUE, CLIM, or Xtk).

2. Coverage

Garnet is designed to handle user interfaces containing graphic objects on which the user can operate with the mouse and keyboard. The objects often represent application data, so the user has the feeling of *directly manipulating* the data since changes made on the screen to the graphics are translated into changes to the data. Similarly, changes the application makes to the data, possibly in response to external events, are reflected on the screen.

Garnet is suitable for applications of the following kinds:

- Box and arrow diagram editors (see Figure 1), like Apple Macintosh MacProject (which helps with project planning).
- Conventional drawing programs such as Apple Macintosh MacDraw.
- Icon manipulation programs like the Macintosh Finder (which let users manipulate files).
- Graphical Programming Languages in which computer programs are constructed using icons and other pictures (a common example is a flowchart).
- Tree and graph editing programs, including semantic networks, neural networks, state transition diagrams, etc.
- Board game user interfaces, such as Chess or Othello (see Figure 2).
- Simulation and process monitoring programs, in which the user interface shows the status of the simulation or process being monitored, and lets the user manipulate it.
- User interface construction tools (Garnet was implemented using itself).
- Some forms of CAD/CAM programs.

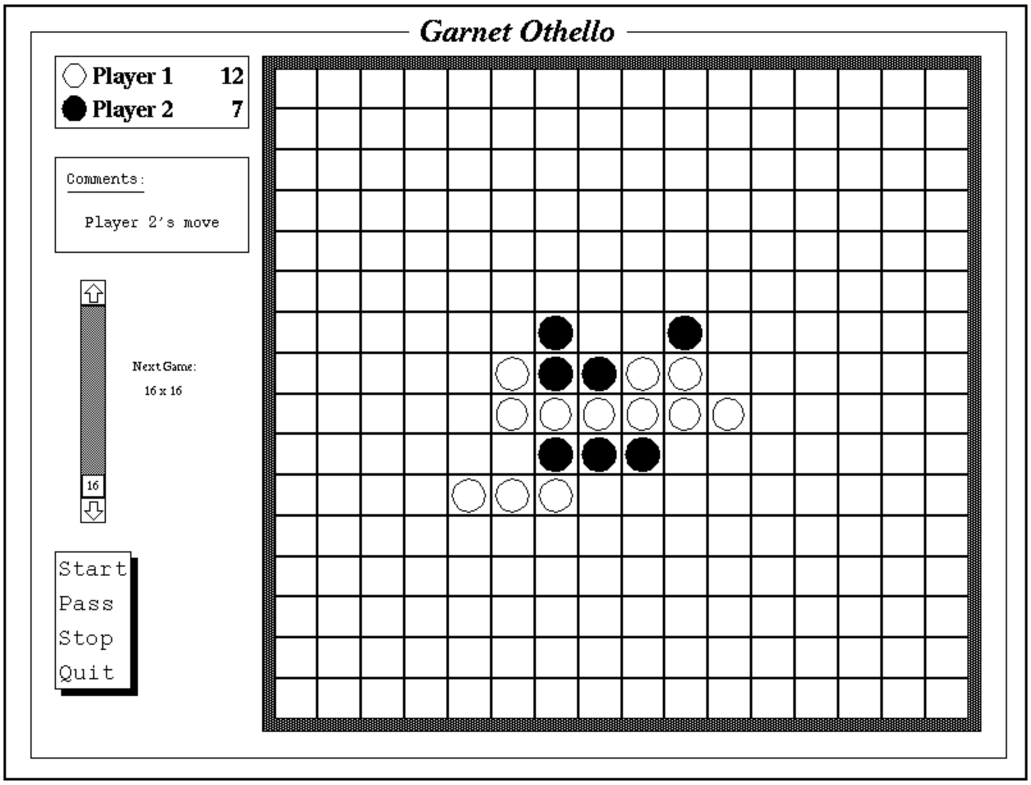


Figure 2: An Othello game created using Garnet.

Garnet does not handle applications with a significant text editing component, but small editable strings that might be used as labels or fields of a table or dialog box are provided.

3. Motivation and Related Work

It is well-known that user interface software is difficult and expensive to implement[7]. Highly-interactive interfaces are among the hardest to create, since they must handle at least two asynchronous input devices (e.g., a mouse and keyboard); real-time feedback; multiple windows; and elaborate, dynamic graphics. Most graphical interfaces today are created using toolkits, which are collections of interaction techniques (sometimes called ‘‘widgets’’ or ‘‘gadgets’’), such as menus, scroll bars, and buttons. Examples of toolkits are 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. In addition, the toolkits often 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 (which are expressed at a low level such as ‘‘the left mouse button went down,’’ ‘‘the mouse is at (30,345),’’ etc.), deciding which operation to perform on objects, and drawing objects using the underlying graphics package (which usually supplies operations such as ‘‘draw-line,’’ ‘‘draw-circle,’’ etc.). Furthermore, it is usually difficult or impossible to modify toolkit items or create new ones.

Higher-level tools such as interface builders and user interface management systems (UIMSs)[7] have not adequately addressed the problems described above. An Interface Builder is a program that lets the designer graphically place user interface components in a window, and thereby create menus, palettes and dialog boxes. Examples include NeXT's Interface Builder, the Prototyper from Smethers Barnes for the Macintosh, and UIMX from Visual Edge for X. These programs only allow pre-programmed interaction techniques to be placed in windows, and then usually allow only a few parameters of these to be set. Often, the designer will type the name of a procedure to be called when the interaction technique is executed. Existing Interface Builders do not allow the interaction techniques *themselves* to be designed or modified in significant ways, and they do not even address application-specific graphics.

A user interface management system (UIMS) is a software system that helps the designer handle the dialog, or sequencing, aspects of the user interface—what happens after each user action. UIMSs have, for the most part, not addressed the creation of toolkit components (menus, scrollbars, etc.) or the specification or management of application-specific graphical objects (the contents of application windows).

The primary influence on the Garnet project is the Peridot UIMS[8]. It is a construction tool that allows toolkit items to be created without programming. Peridot lets non-programmers create many types of interaction techniques, including most kinds of menus, property sheets, buttons, scroll bars, percent-done progress indicators, sliders, iconic and title line controls for windows, and many others. Like Garnet, Peridot uses constraints. However, Peridot lacks a programming interface, there is no way to use existing toolkit items, and it cannot create application-specific graphic objects.

Constraints have been used by many systems including Thinglab[9] and Apogee[10]. The box on page 23 discusses constraints in more detail.

The Jade dialog editor, which automatically constructs dialog boxes from high-level specifications (see section 7), was influenced by the Interactive Transaction System[11].

4. The Garnet Toolkit

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

Figure 3: The structure of the Garnet system.

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

1. an object-oriented programming system, called KR,
2. a constraint system,
3. a system for handling input, called Interactors,
4. a graphical-object system, called Opal, and
5. a collection of widgets.

In the terminology of the X toolkit, the first four parts are the Garnet Toolkit ''intrinsics'' (the mechanism that supports the implementation) and the fifth is the Garnet ‘‘widget set’’ (a collection of menus, scroll bars, etc. with a pre-specified look-and-feel).

4.1 Object-Oriented Programming System

The Garnet object-oriented programming system is called KR. It supports a prototype-instance model for objects, rather than the conventional class-instance model used by Smalltalk and C++. In a prototype-instance model, there is no distinction between instances and classes; any instance can serve as 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, there is no distinction between data and method slots in KR. 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 (e.g., `: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 inherit defaults for, and which it wants to override. In fact, the number of slots of an object 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 feature is that it is easy to create slots in objects to hold local data. For example, if the color of a rectangle represents an application temperature, the application can create an instance of a rectangle and create in it a slot called `:temperature`. A disadvantage of this flexibility is that it is impossible to perform compile-time or even run-time checking of slot accessing and setting, 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 how inheritance works, assume a top-level rectangle prototype has slots containing values for the left, top, width, height, and color of the rectangle (see Figure 4). When an instance is created using that rectangle as a prototype, it will typically override some of the values, but it is not necessary to do so. A programmer could create an instance with a particular color and size, and then create more instances of *that* rectangle (see Figure 4). If a value of a slot in a prototype is changed, all instances of that prototype that do not override that slot immediately inherit the new value. Similarly, if a slot of an instance is removed, the corresponding slot of its prototype will start to be used instead, if it exists. All objects (prototypes and instances) can be displayed on the screen.

Note that this allows the methods that implement messages sent to the objects to change dynamically, which is not possible in conventional object systems like Smalltalk. In Garnet, it is only necessary to assign a new procedure into the appropriate slot of an object, and the new method will be used subsequently.

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` is changed, then the lefts of `MyRectangle` and `Rect1` immediately change also. Even though they may be serving as prototypes for other objects, all of these are ‘‘real’’ objects in that they can be displayed on the screen.

The advantages of the prototype-instance model are that it is much more dynamic and flexible than the familiar class-instance model. A high-level tool, such as the Garnet interface builder, can display a prototype on the screen, and allow the user to edit it. These edits are then automatically reflected in all instances of that prototype. For example, the designer might be changing the standard look-and-feel of the menu prototype, and immediately all menus in the system would change accordingly. In a class-instance model, changing the class structure when there are existing instances is either very expensive or else causes the instances to be invalid.

A potential disadvantage of a prototype-instance model is speed. Since slots must be inherited from the prototypes if they are not present in an object, getting the value from a slot could require a search up the entire inheritance hierarchy. This problem is alleviated in Garnet by keeping local caches of inherited values. Another efficiency problem is that, since new slots can be added to objects at any time, a fixed storage scheme like a structure or record cannot be used, but 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 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 accessor function takes about twice as long in PCL (Portable Common Loops, which is an implementation of CLOS) (49.8 microseconds), than in KR (27.3 microseconds). To create an instance in CLOS (16160 microseconds) takes about 15 times longer than in Garnet (1117 microseconds).

4.2 Constraint System

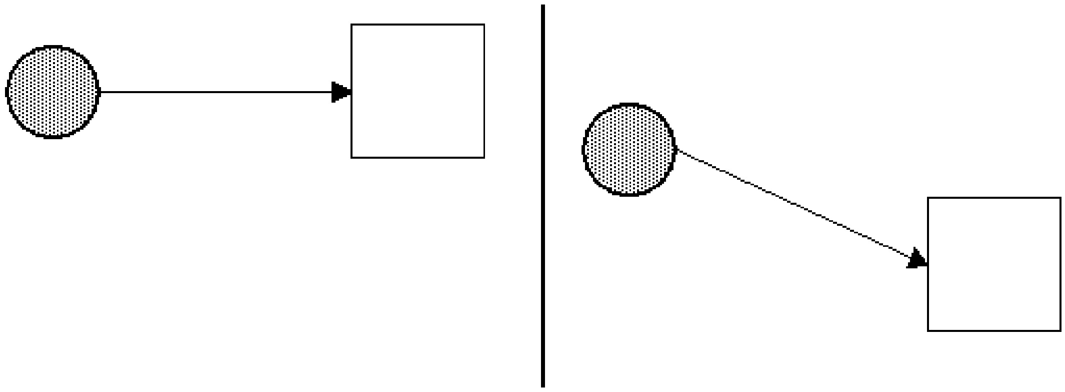
A *constraint* is a relationship among objects that is maintained when any of the objects changes. Constraints are a natural way to express common relationships in graphical user interfaces. For example, in an editor that supports boxes attached by arrows, the user interface designer could specify a constraint that the arrows stay attached to the boxes. Then, when the boxes are moved by a program or the mouse, the system will automatically move the arrows as well. The box on page 23 discusses in more detail why constraints are useful and how they are implemented.

An early version of the constraints in Garnet was Coral[1]. This implementation was abandoned because it was too slow and not sufficiently flexible. Currently, constraints are integrated with the KR object system.

Constraints in Garnet are arbitrary Common Lisp expressions, which are stored in slots of objects. When a program accesses a slot, it cannot tell whether the slot contains a simple value like a number, or a constraint that calculates the value. Inside constraints, references to slots of other objects use a special form: `(gv other-object slot-name)`, where `gv` stands for ''get value.'' Whenever the referenced slot of the other object changes, the formula is re-evaluated. For example, to get an arrow to stay attached to two objects, the code in Figure 5 could be used.

These formulas are ''one-way'' constraints, which means that if the other object changes, the object with the formula is re-evaluated, but not vice-versa. For example, in Figure 5, when the circle or box is moved, the line will move, but if the line is moved, the circle or box will not be moved. 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.

More powerful ''multi-way'' constraints have been used in other systems (see box on page 23), and may be added to Garnet in the future. The primary advantage of the current scheme, however, is that



```
(create-instance 'myline arrow-line
                 (:x1 (formula (gv circle1 :right)))
                 (:y1 (formula (gv circle1 :center-y)))
                 (:x2 (formula (gv box1 :left)))
                 (:y2 (formula (gv box1 :center-y))))
```

Figure 5: The line stays attached to the box and circle even when they are moved. At the bottom is the code used to define the constraints on the line.

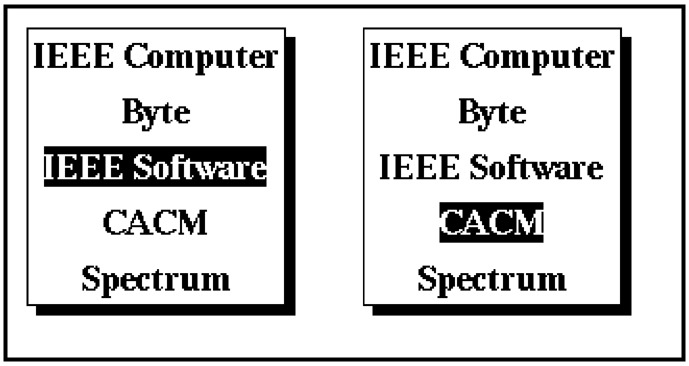
constraint evaluation is very efficient. Garnet can re-evaluate over 3500 constraints per second on the IBM/RT implementation, which means that many objects with dozens of constraints can be updated in real-time as objects are dragged with the mouse. Of course, Garnet ensures that formulas whose values do not change are not re-evaluated, so systems can contain many thousands of constraints.

An interesting and novel feature of the constraints in Garnet 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 will hold the current object that the feedback should appear over. Whenever this slot is changed, Garnet will automatically re-evaluate the formulas that depend on the slot, thus causing the feedback object to move. In Garnet, this is called a ''link.'' To make this easy, the `gv` function can be passed a list of slots to use as indirect references. For example, if `(gv :SELF :other-obj :left)` appeared in a formula, this means to look in the `:other-obj` slot of this object. The contents of this slot will be another object. Go to *that* object and get its left. Since this form is very common, `(gvl ...)` can be used instead of `(gv :SELF ...)`.

As an example of the use of links, in Figure 6, the reverse video rectangle will move whenever the value of the `:obj-over` slot is changed. It is this mechanism that allows the specification of the input handling to be independent of the graphics. For example, the interactor object that handles menu behavior (described below) simply sets the `:obj-over` slot to the object that the mouse is over, and the constraints ensure that the graphics that handle feedback are changed appropriately. The interactor does not need to know anything about the appearance of the graphics.

The use of constraints in Garnet is not limited to only the position and sizes of graphical objects. Because any slot of any object can contain constraints, they 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 (gvl :which-button) :rightdown)))
        ;grow if right button is pressed, otherwise move
```




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

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

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.

4.3 Graphical-Object System

The Garnet graphical object system is called Opal, and is designed to make it easy to create and edit graphical objects. To this end, Opal provides default values for all properties of objects, so that 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, it is only necessary to 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 property of an object can be specified using formulas instead of numbers, as shown in the code example of Figure 5.

Another important feature of Opal is that it automatically handles object drawing and erasing. If any property of an object is changed, Opal automatically refreshes the screen and redraws that object. If that object overlaps others on the screen, Opal ensures that these are also redrawn correctly (see Figure 7). In addition, if the modifications cause other objects to change due to constraints, these other objects are also redrawn automatically. The algorithm used by Opal tries to minimize the number of objects that are erased and redrawn, rather than simply redrawing all the objects. 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 redrew all the objects.

Because Opal knows where all objects are, it can also handle window refresh (when the window becomes uncovered). Clients of Opal never have to worry about the X refresh events. In fact, Opal and the Interactors completely hide all X functions.

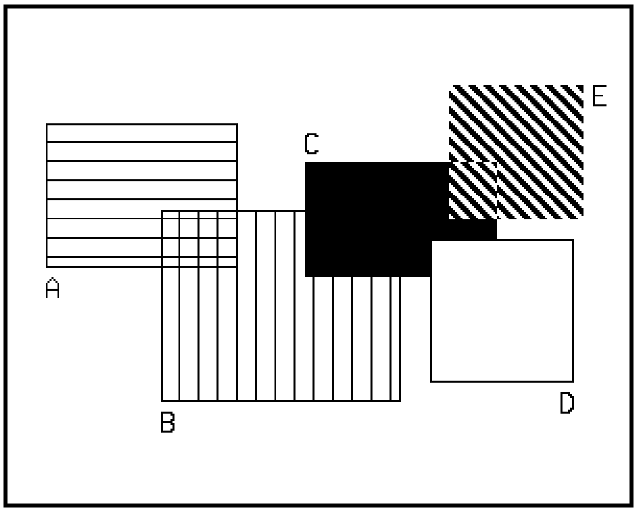


Figure 7: Opal allows graphics to overlap opaquely, and automatically refreshes damaged parts. If C is erased, B, D and E will need to be redrawn. Opal uses a clipping region so that only the parts of B, E, and D that overlap C are affected, and so A does not have to be redrawn. B is drawn with the OR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

To make an object appear in Opal, it is only necessary to add it to an Opal window. To make it disappear, the object is removed from the window. To change an object's color or size, the appropriate slots are set. 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 the 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. This means that an instance is made for each of the components of the Aggregate, as well as for the Aggregate itself (see Figure 8). Changes to the Aggregate are immediately reflected in all instances, including when components are added or deleted from the prototype. In this case, the corresponding components are immediately added or deleted from all instances. Previous implementations of the prototype-instance model have not supported changing of the *structure* of instances in this way.

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

Of course, each instance can override any of the slots. This makes it extremely easy to make prototypes for complex, composite objects like menus and scroll bars. The prototype contains example values for slots like the strings in the menu, and the instances override these values.

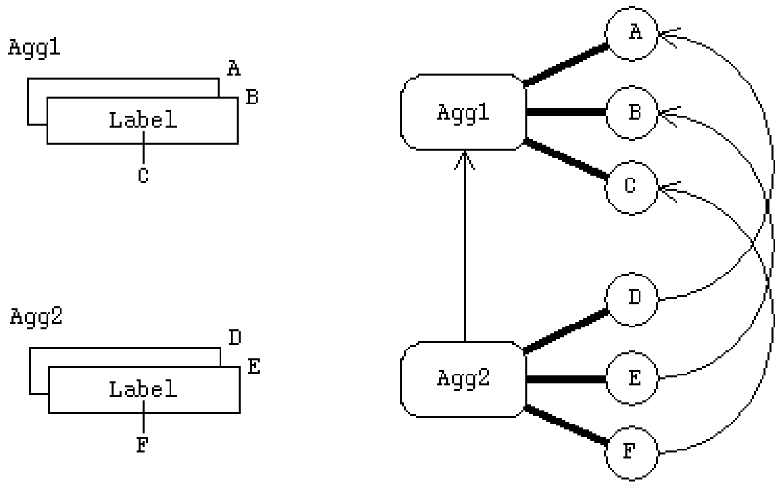


Figure 8: `A`, `B` and `C` are components of the aggregate `Agg1`. When an instance is made of `Agg1`, instances are made of each of its components automatically. The dark lines are component links, and the arrow lines are instance links. If `Agg1` or any of its components are changed, `Agg2` and its components would be changed automatically.

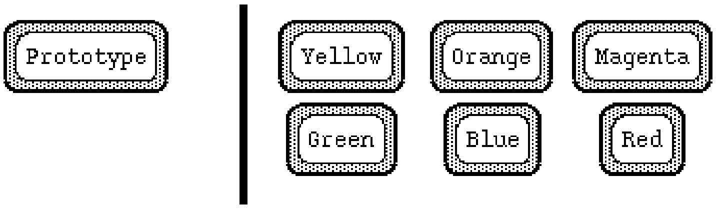
A special form of Aggregate is an ‘‘AggreList,’’ which is used to lay out a list or table of elements. In an AggreList, there is a single prototype for all the elements, and each element uses a different value for some value of that prototype. For example, menus are usually implemented as an AggreList that uses a single prototype for the items, but a different string value displayed in each item. In Figure 9-a, the prototype is an Aggregate containing the 2 rounded rectangles (one grey and one white) and the example string. AggreLists support displaying the items horizontally, vertically, and in multiple rows, or else the programmer can specify a layout (as was done for the labels in the gauge of Figure 12-f).

4.4 Input Handling

One of the most difficult tasks when creating highly-interactive user interfaces is handling the mouse, keyboard, and other input devices. Typically, window managers and user interface toolkits only provide a stream of device-dependent mouse positions and keyboard events and 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[2]. The observation that makes this feasible is that there are relatively few distinct *behaviors* employed in user interfaces. For example, although the graphics can vary significantly and the specific mouse buttons used may change, all menus operate in essentially the same manner. Another example is the way that objects move around when being dragged with the mouse. The Interactors capture these common behaviors in a central place while still being highly customizable by application programs.

Other advantages of the interactors are that:

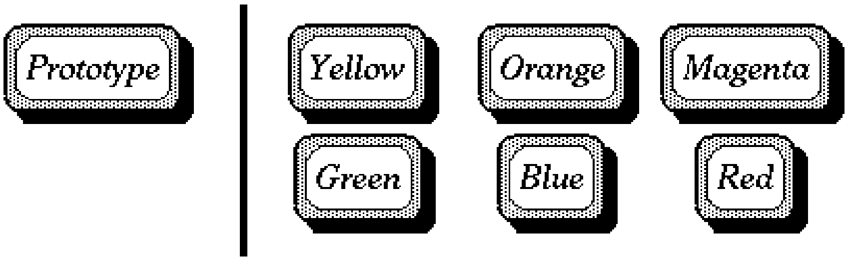
- They are entirely ‘‘look’’ independent; any graphics can be attached to a particular ‘‘feel’’ (see Figures 10 and 11).
- They allow the details of the behavior of objects to be separated from the application and from the graphics, which has long been a goal of user interface software design.



(a) (b)

```
(create-instance 'colorlist AggreList
                 (:left 150)(:top 10)
                 (:Item-Prototype My-Roundtangle-Prototype)
                 (:Items '("Yellow" "Orange" "Magenta" "Green" "Blue" "Red"))
                 (:direction :horizontal)
                 (:h-align :center) ;Center the objects in the field.
                 (:rank-margin 3) ;Go to next line after 3rd object.
                 (:fixed-width-p T)) ;All fields have same width.
```

(c)



(d) (e)

Figure 9: A prototype for the menu items (a), and a two-dimensional AggreList 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 (d) to add a drop shadow and change the font, all instances immediately inherit the new values and structure (e).

- They support multiple input devices operating in parallel.
- Different applications can be running in separate windows in the same Lisp process and same address space, and the interactors will allow the user to operate on any of them, thereby supporting a form of multi-processing.
- All of the complexities of X graphics and event handling are hidden by Opal and the Interactors package. This makes Garnet much easier to use than X, and may also allow Garnet to be ported to other graphics packages, such as Macintosh QuickDraw or Display Postscript, without requiring significant changes to applications written using Garnet.

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

`Menu-Interactor`: for choosing one or more from a set of items, or for a single, stand-alone button.

`Move-Grow-Interactor`: to move or change the size of an object or one of a set of objects using the mouse. This interactor can be used for one-dimensional or two-dimensional scroll bars, horizontal and vertical gauges, and for moving or growing application objects in a graphics

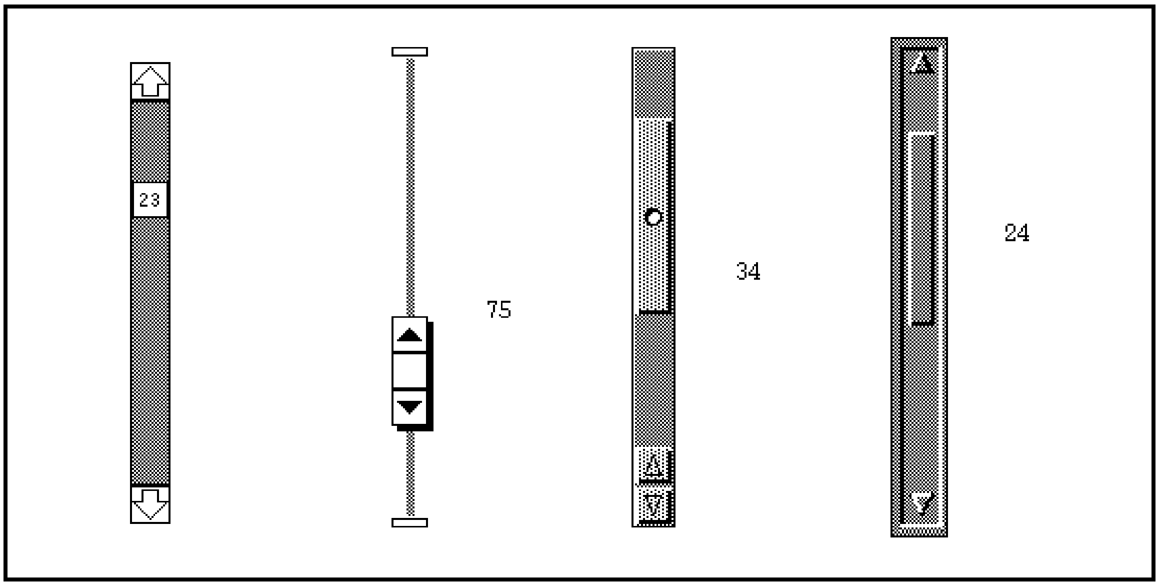


Figure 10: The same type of interactor can handle different graphic looks. Here, move-grow-interactors for the indicators and menu-interactors for the arrow buttons are handling scroll bars that look like those in the Macintosh, OpenLook, NeXT and OSF/Motif.

editor.

`New-Point-Interactor`: to enter 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`: to calculate the angle that the mouse moves around some point. It can be used for circular gauges or for rotating objects.

`Trace-Interactor`: to get all of the points the mouse goes through between start and end events, as is needed for free-hand drawing.

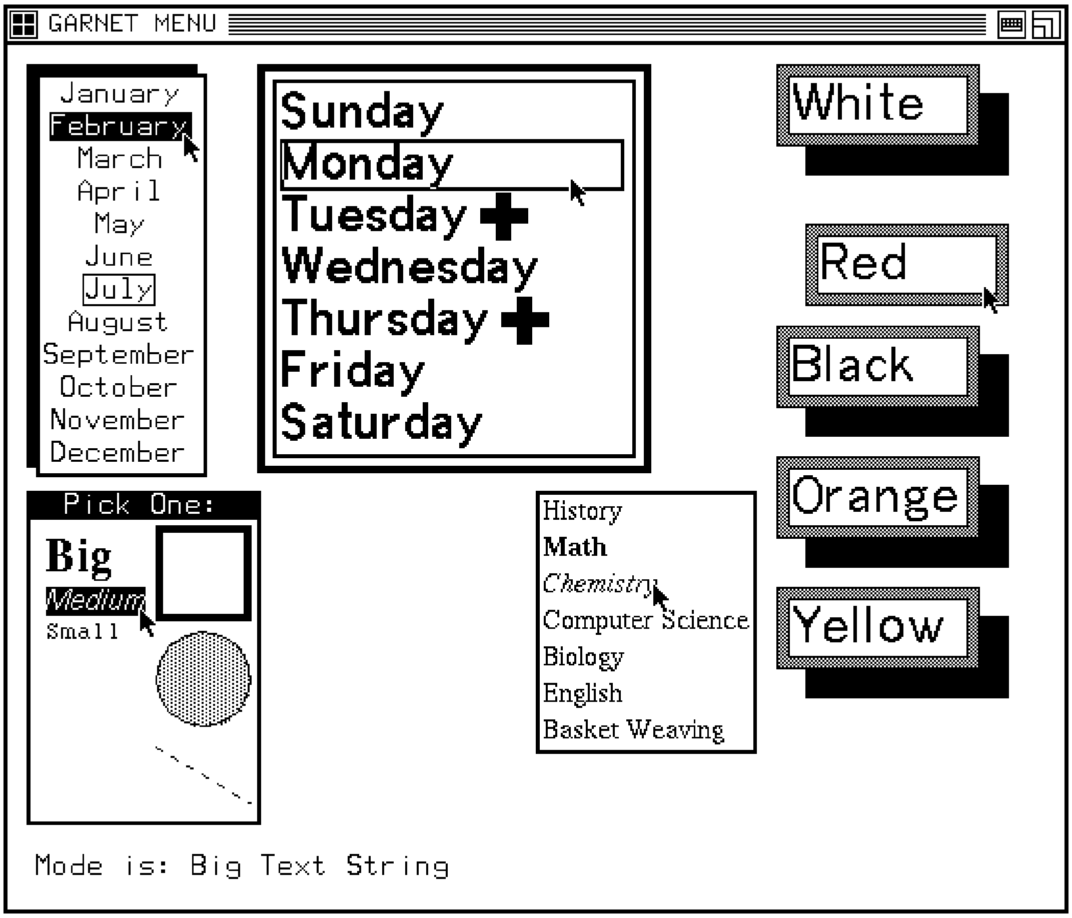
`Text-String-Interactor`: to input a small (optionally multi-line) string of text.

Each interactor is parameterized in various ways, so the programmer can control the mouse or keyboard events that cause it to start and stop, and the optional application procedures to be called on completion. The most significant parameters, however, are the objects that are used as the places where the interactor should operate, and the (optional) objects that will handle feedback. Each type of interactor has a well-defined protocol with which it controls the graphics. For example, the menu interactor sets the `:obj-over` slot of the feedback object. Therefore, to turn the list of items in Figure 9-b into a menu using the feedback object of Figure 6, the following code is all that is needed:

```
(create-instance 'color-selector menu-interactor
        (:start-where '(:element-of colorlist)) ;Colorlist is defined in Figure 9-c.
        (: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 move-grow ‘‘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 may change in response to the mouse. If the `:feedback-obj` is NIL, 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 11-c has constraints that change the position of the object under the mouse based on whether the value of `:interim-selected` is T or NIL, and the menu of Figure 11-e

(a) (b) (c)



(d) (e)

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, and the final selection is bold.

decides whether to 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 will be 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 one to move based on where the mouse 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, whenever a new style of object is desired, it is usually necessary to write code for all three parts. In Garnet, programmers virtually 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 supply appropriate parameters to achieve the desired behavior.

New types of interactors would only be needed in Garnet for radically different types of behaviors, such as gesture recognition or new kinds of physical input devices (such as a 3-D joystick). Constraints in Garnet are used to connect the Models, Views, and Controllers together.

4.5 Gadgets

On top of Opal and the Interactors is a collection of interaction techniques which provide a starting point for applications. There are gadgets (also called ‘‘widgets’’) for menus, scroll bars, buttons, gauges, etc. Figure 12 shows some of the gadgets supplied with Garnet. These can be used by applications that do not want to create their own look-and-feel. Most of the gadgets have a number of parameters so that the designer can vary many aspects of the appearance and behavior.

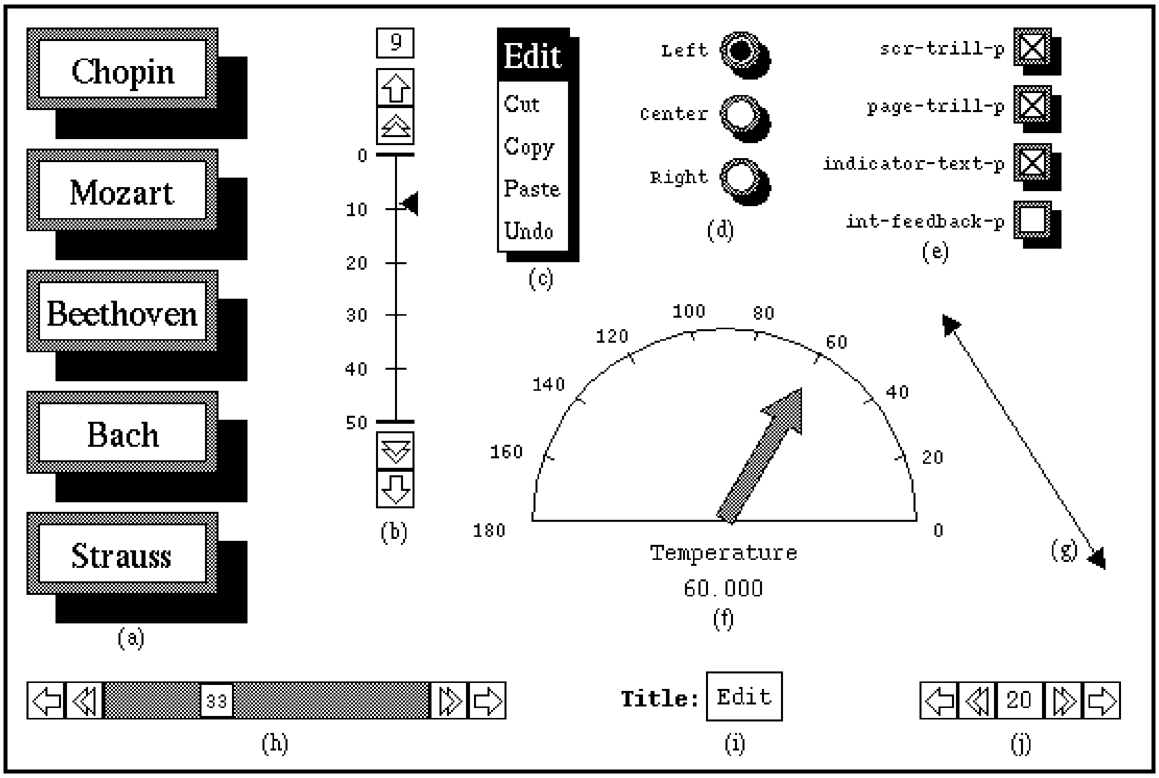


Figure 12: Some of the gadgets provided by Garnet: (a) floating buttons, (b) a number slider, (c) a menu, (d) floating radio buttons, (e) floating check-boxes, (f) a semi-circular gauge, (g) an arrow-line, (h) a scroll bar, (i) a labeled text entry field, and (j) a number entry field.

In some toolkits, such as Xtk and InterViews[6], each gadget is a window. This significantly limits what gadgets can do. In Garnet, however, gadgets are not windows, so, for example, there is a gadget that is a line with an attached arrowhead (Figure 12-g). An even more sophisticated gadget in Garnet handles object selection (see Figure 13). It displays ‘‘selection handles’’ around the selected object, and then allows the user to move or grow the object by pressing on the handles. Therefore, to support graphical object selection, a Garnet application need only create an instance of an object selection gadget (assuming that the Garnet look-and-feel is acceptable).

Since it is very easy to create new gadgets using Garnet, we expect that many applications will create

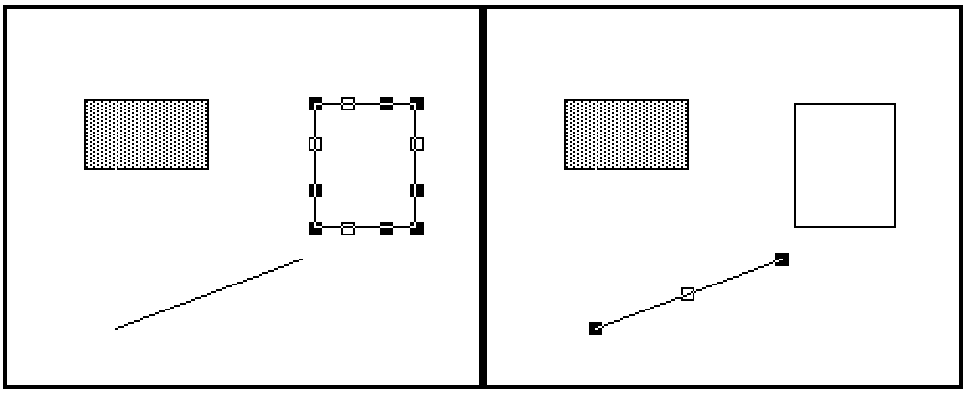


Figure 13: The graphics selection gadget shows control handles around the selected object. Pressing on a white handle moves the object, and pressing on a black one changes the object's size. When a line is selected, only three control points are shown: the black ones change the end point and the white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

their own rather than use these presupplied ones. By using Lapidary or by programming using Opal and Interactors, new styles of menus and buttons can be created in minutes.

5. Debugging Tools

Although defining user interfaces using constraints and interactors is quite natural and effective, bugs in the code can be very difficult to find. This is because the effects of a bug are not local; constraints can cause a bad value in one place to cause many different objects to have bad values. Therefore, Garnet provides a large and growing number of debugging tools to help with program implementation. These provide tracing mechanisms to show how a certain slot got its current value, where objects are on the screen or why they are not visible, what happens when the mouse or keyboard is used, and extensive checking for legal values of graphics slots. In addition, there are convenient ways to inspect objects, including browsers (see Figure 14).

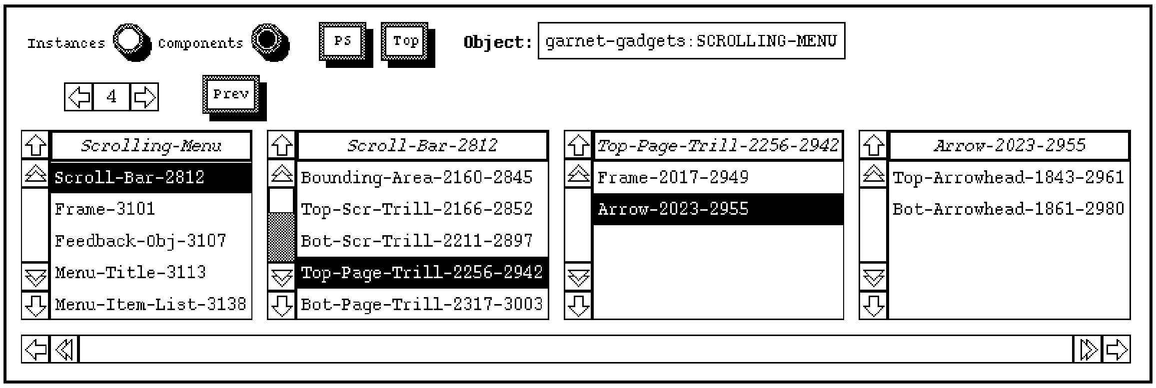


Figure 14: A browser that will either show the components of an aggregate or the instances of a prototype. This was created using the Garnet Toolkit, and can be used to debug Garnet code.

6. Garnet Interface Builder

On top of the Garnet Toolkit layer are a number of tools to make creating user interfaces significantly easier. The most important is the Lapidary interface builder[3]. Lapidary provides a graphical front end to most of the underlying Garnet Toolkit features, so that *all* graphical aspects of programs can be specified pictorially. In addition, the behavior of these objects at run-time can be specified using dialog boxes and by demonstration.

In particular, Lapidary allows the designer, who does not have to be a programmer, to draw pictures of application-specific graphical objects which will be created and maintained at run-time by the application. This includes the graphical entities that the end user will manipulate (such as the components of the picture), the feedback that shows 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 hair-line boxes to show where an object is being dragged). The designer creates prototypes of the objects in Lapidary, and then the application program creates instances of these as needed.

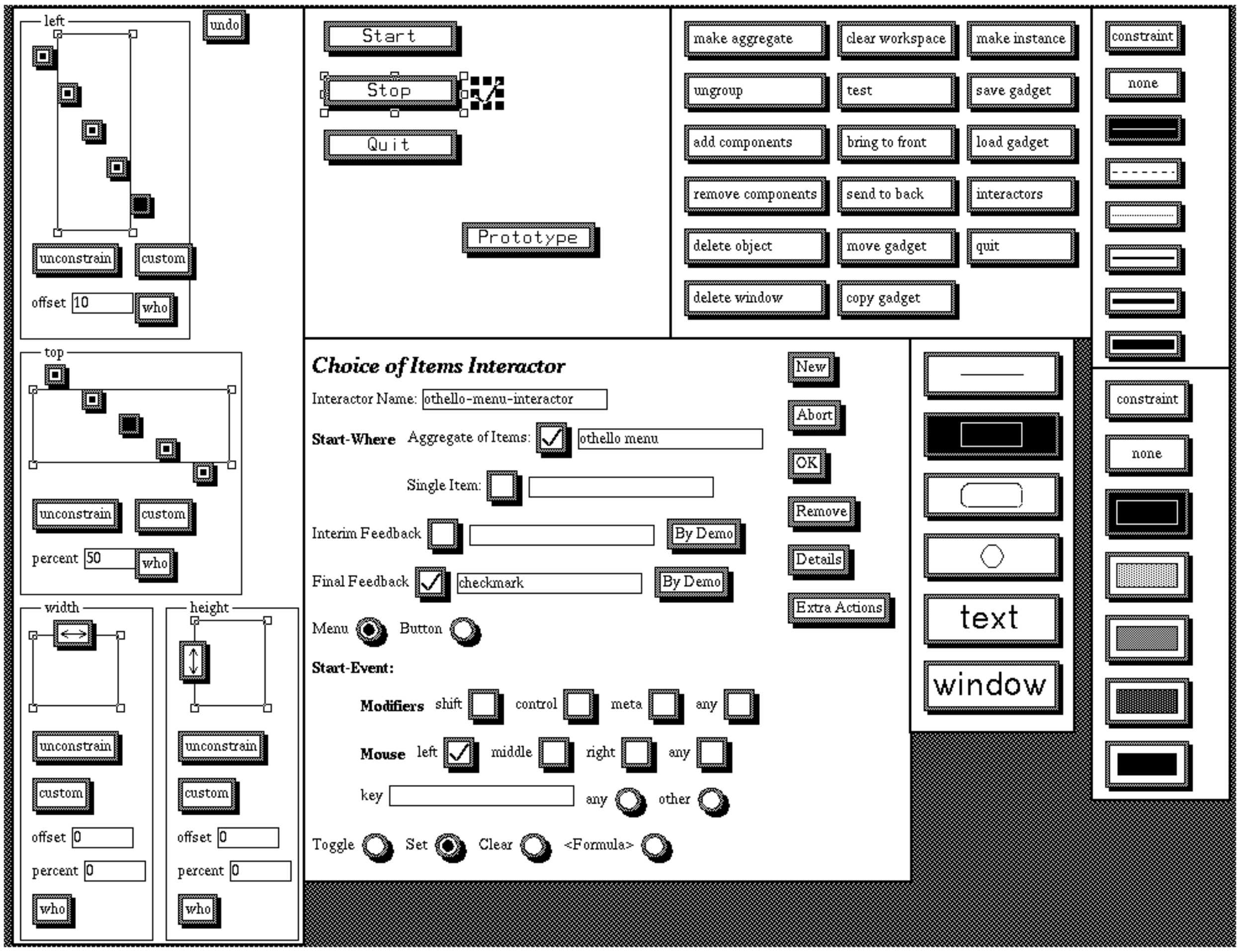
In addition, Lapidary supports the construction and use of interaction techniques, such as menus, scroll bars, buttons and icons. Lapidary therefore supports both *using* a pre-defined library of widgets, and *defining* new widgets with a unique ‘‘look and feel’’ (and these can even be combined in the same application). The run-time behavior of all these objects can be specified in a straightforward way using constraints and abstract descriptions of the interactive response to the input devices. Lapidary generalizes from the specific example pictures to allow the graphics and behaviors to be specified by demonstration.

The designer can specify the behavior of objects in various ways in Lapidary. Graphical constraints 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 they refer to whatever graphical object the mouse is over. For example, the check mark is constrained to the middle button in Figure 15-b, but Lapidary generalizes this constraint to use a variable (like the `:obj-over` slot of Figure 6) so the check-mark will appear over any of the buttons.

Also, if an object should change based on some user action, the designer can specify this *by demonstration*. First, one state is drawn, and then another state, and Lapidary will automatically construct the constraints to change the object between the two states. This can be used to define the menus of Figures 11-c and 11-e.

As another example of how Lapidary works, the following is a brief overview of how a user interface designer might create the boxes and arrows for a graph editor. First, the user interface designer would draw a picture of the boxes, say using rounded-rectangles (Figure 16-a). Then, an example text label would be created and centered at the top using the iconic constraint menus (Figure 16-b). This will serve as the prototype for the boxes the application will create, so the designer selects the entire group and gives a command to save the group as a named prototype. Lapidary asks the designer what the parameters to the object should be, and the designer specifies the position and size of the box and the label string. Lapidary will then write this prototype to a file, and also define it in memory (in case the application is running concurrently with Lapidary). Next, the designer creates a copy of the box, and then draws an arrow-line (Figure 16-c). The line constraint menu is then used to specify that the ends of the line should be centered in the boxes (Figure 16-d). Then, the line by itself is selected and saved as a prototype. Lapidary notices that there are constraints to objects that are *not* being saved, and 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 boxes that go into the `from-obj` and `to-obj` slots, and new lines will appear in the application window. The application knows nothing about the graphics used for the lines, and the designer did not have to

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



(e) (f) (g)

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

write any graphics code. This entire design session takes about four minutes.

7. Automatic Dialog Box and Menu Creation

The motivation for the Jade dialog box creation system is that sometimes it is easier to list the contents of a dialog box or menu, rather than to meticulously draw it. Jade automatically creates an attractively laid out dialog box or menu from a specification of its contents[4]. In addition to being simple to use, the specification passed to Jade has the additional advantage of being look-and-feel independent. The textual specification of the contents simply lists the string labels to appear in the dialog box along with the type of input required (e.g., choice of one of a set, a number in a range, etc.), and the particular look-and-feel to use (e.g., Macintosh, Motif, Garnet-standard, etc.). The specification has a Lisp-like syntax. From this, Jade will choose the correct interaction techniques, which themselves are designed using Lapidary. In addition, 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 dialog box go away (e.g., ‘‘OK,’’ ‘‘CANCEL’’) will be at the right for a Macintosh-like dialog box, and at the top for a Xerox-Star-like one. The dialog box in Figure 15-e was created automatically by Jade.

The heuristics for placing objects in dialog boxes attempt to create an attractive display and are based on principles of good user interface design. If the designer is not happy with the result, however, the resulting dialog box can be read into Lapidary and edited. Decorations, such as extra rectangles and lines, can be added, and parts can be moved around. These changes are saved in an exceptions file, so they can be re-applied even if the original textual specification is edited and the dialog box regenerated.

8. Spreadsheet for Constraint Specification

Although many of the desired constraints can be specified using the iconic menus in Lapidary, the designer occasionally needs more powerful constraints. The C32 spreadsheet program[13] in Garnet allows arbitrary Lisp constraint expressions to be entered. It provides many of the advantages for graphics that financial spreadsheets provide for business. In particular, C32 provides the ability to monitor and debug interfaces by watching values in the spreadsheet while the user interface is running, extensible menus of commands, 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.

9. Status and Future Work

Garnet is under active development at Carnegie Mellon University. The Garnet Toolkit is operational, and there are many local and external users. As of this writing, over 200 companies and universities all over the world have requested a license. It is available for free from CMU, but you need to have a license. If you are interested in using Garnet, please contact the first author.

Currently, we are working on increasing the functionality and performance of the Garnet Toolkit and finding new ways to support the *entire* application user interface design and implementation process. Since the Toolkit is now available, we hope that it will get wide distribution and use in the Common Lisp community.

Lapidary is working but not yet in a releasable state. Jade is partially working, and has been used to generate the Lapidary dialog boxes. The C32 system is just now being implemented.

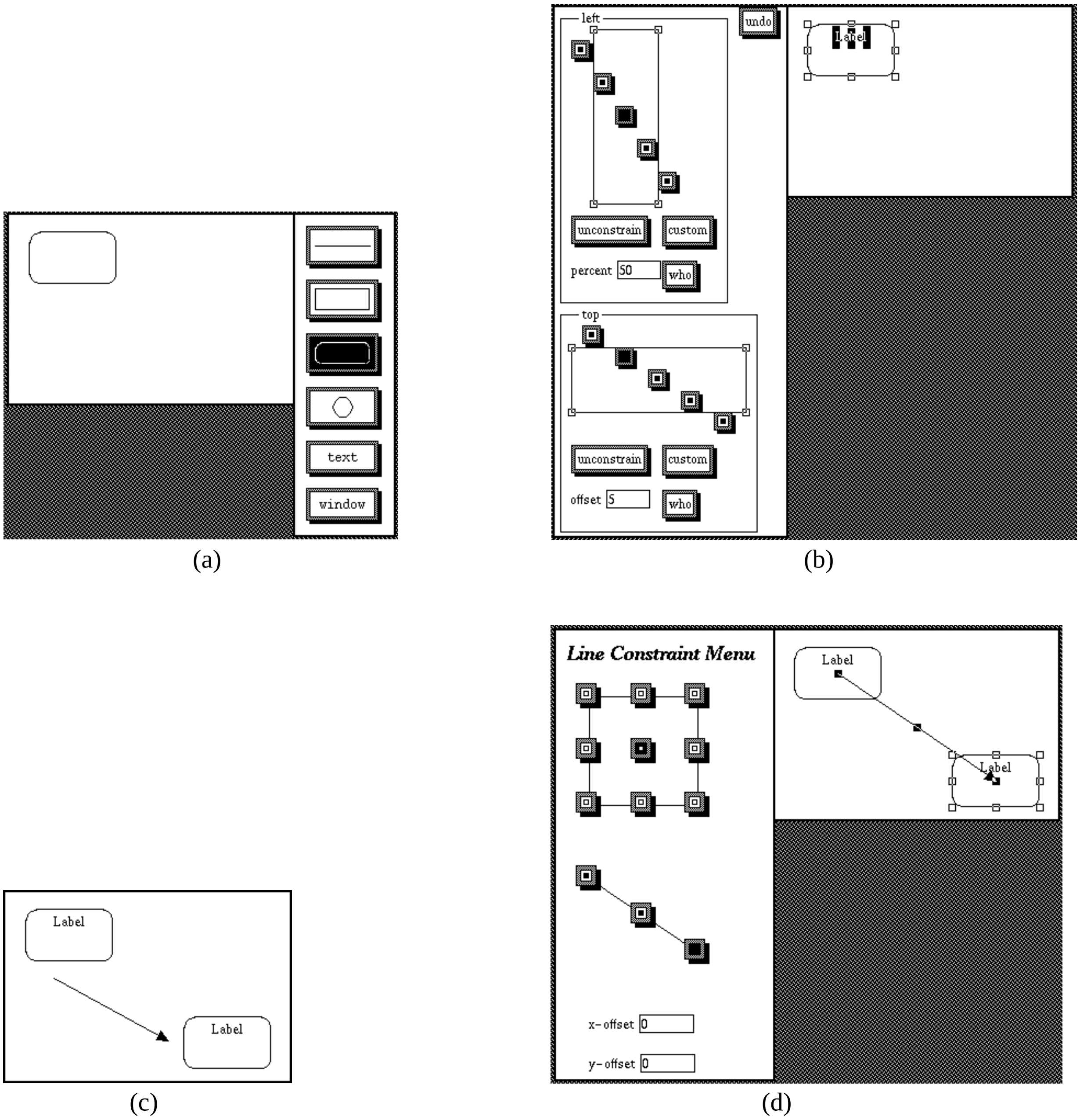


Figure 16: A sequence of snapshots while creating application-specific graphics for a graph editor. (a) Drawing an example box, (b) centering the text label using the iconic menus, (c) drawing the arrow line, and (d) constraining the second endpoint of the line to be centered in a box.

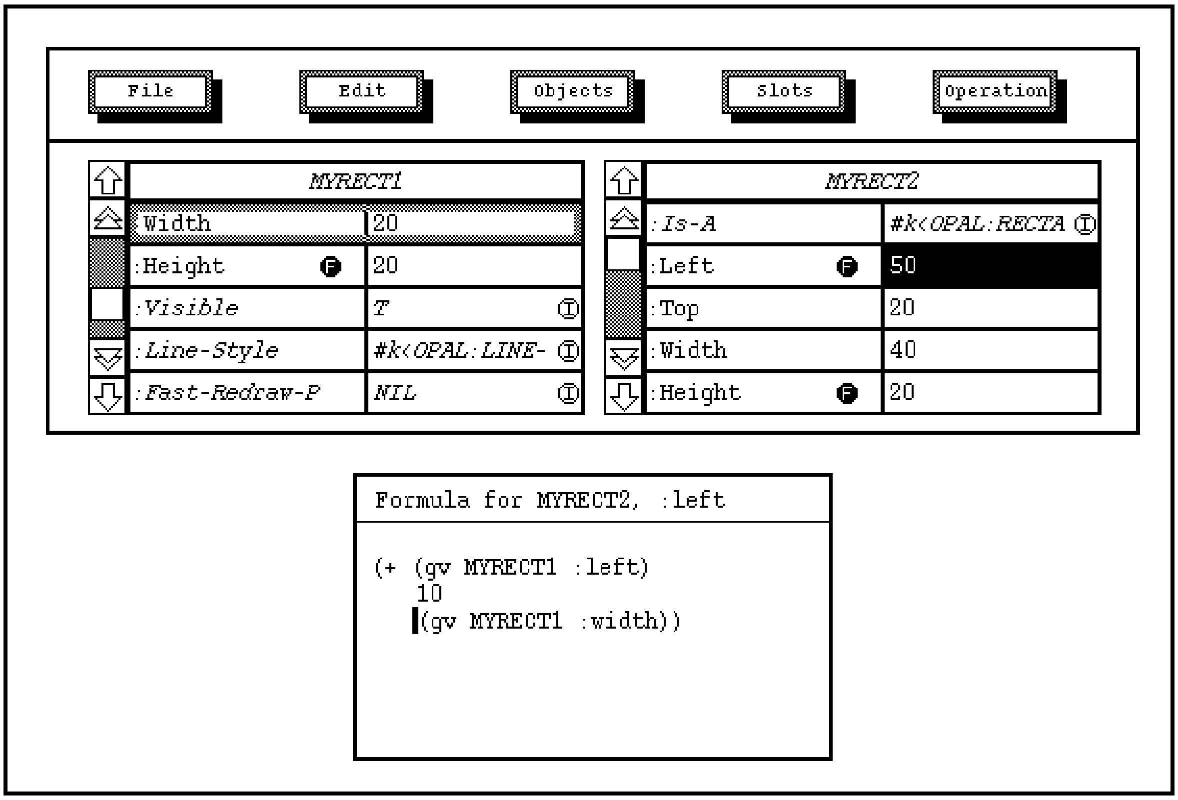


Figure 17: An example of the Garnet C32 spreadsheet viewing two objects. At the top of the window are a set of pull-down menus. Each object has its own set of columns. The slot names of the objects are shown at the left of each column, along with the current value. If the value is computed by a formula, an icon with an ''F'' is shown. Inherited values are shown in italics and are 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 on an object in a graphics window, a reference to that object is inserted into the formula. Here, a reference to the `:Width` of `MYRECT1` has just been inserted, and that slot is outlined in gray.

We will be working to have more of the interface specified by demonstration, rather than through dialog boxes and coding. For example, Lapidary could infer graphical constraints and mouse dependencies from the drawing, in a manner similar to Peridot[8]. In addition, we 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 common operations found in most systems that allow the user to create and manipulate graphical objects. This might include support for a palette of object types, selection of objects, changing their size and position and other properties, deleting objects, automatic layout for objects as a list, table, graph or tree, saving and restoring objects to a file, printing, undo, help, etc. For many programs, this subsystem will provide most of the standard functionality, and it will only be necessary to specify the application-specific parts.

10. Conclusion

Although Garnet has only been working for a short time, it has already demonstrated that it makes the creation of graphical, highly-interactive user interfaces significantly easier. For example, in an informal experiment, creating a simple graphical editor like that in Figure 1 took only 2 to 4 hours with the Garnet Toolkit, but 10 to 20 hours using other toolkits such as Apple's MacApp and Sun's OpenLook.

Garnet is one of the few systems that supports the creation and exploration of various looks-and-feels for user interfaces. It is also the only system with an emphasis on allowing the behavior of objects to be easily specified, often by demonstration without programming. The use of constraints and automatic refresh for graphical objects has proven to be very useful and sufficiently efficient to support the desired interfaces. The encapsulation of the interactive behaviors makes it much easier to have the objects respond to input devices. The Lapidary interface builder allows most of the user interface to be specified graphically and by demonstration without programming, and Jade is the only look-and-feel-independent dialog box creation system. The C32 spreadsheet helps designers create complex constraints. Taken all together, these components make Garnet an exciting and innovative system that is extending the state of the art in user interface software, while still being useful for creating user interfaces today.

Acknowledgements

In addition to the authors, the others who have have helped design and implement Garnet are Pedro Szekely, Jake Kolojejchick, and Lynn Baumeister. Amy Moormann Zaremski 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 paper.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, Amendment 20, under contract F33615-87-C-1499, monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), Wright-Patterson AFB, Ohio 45433-6543. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

References

1. Pedro A. Szekely and Brad A. Myers, ‘‘A User Interface Toolkit Based on Graphical Objects and Constraints’’, *Sigplan Notices*, Vol. 23, No. 11, November 1988, pp. 36-45, ACM Conference on Object-Oriented Programming; Systems Languages and Applications; OOPSLA'88

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

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

4. Brad Vander Zanden and Brad A. Myers, ‘‘Automatic, Look-and-Feel Independent Dialog Creation for Graphical User Interfaces’’, *Human Factors in Computing Systems*, Proceedings SIGCHI'90, Seattle, WA, April 1990, pp. 27-34.

5. Brad A. Myers, Dario Giuse, Roger B. Dannenberg, Brad Vander Zanden, David Kosbie, Philippe Marchal, Ed Pervin, Andrew Mickish, James A. Landay, Richard McDaniel, and Don Hopkins,

''The Garnet Reference Manuals: Revised for Version 2.1'', Tech. report CMU-CS-90-117-R3, Carnegie Mellon University Computer Science Department, November 1992.

6. Mark A. Linton, John M. Vlissides and Paul R. Calder, ''Composing user interfaces with InterViews'', *IEEE Computer*, Vol. 22, No. 2, February 1989, pp. 8-22.

7. Brad A. Myers, ''User Interface Tools: Introduction and Survey'', *IEEE Software*, Vol. 6, No. 1, January 1989, pp. 15-23.

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

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

10. 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, October 1988, pp. 167-178.

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

12. Glenn E. Krasner and Stephen T. Pope, ''A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 system'', *Journal of Object Oriented Programming*, Vol. 1, No. 3, August 1988, pp. 26-49.

13. 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.

A Constraints Primer

by
Brad Vander Zanden and Brad A. Myers

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

Constraints provide a convenient way of specifying these relationships and having them *automatically* maintained at run-time by a constraint solver. In contrast, a conventional programming language requires the application to both specify the relationships *and* do all the bookkeeping necessary to maintain these relationships. For example, suppose that an arrow must always stay connected to the center of a box and a label on the arrow must always stay centered on the arrow. In constraint programming, whenever the application changes the position or size of the box, the constraint solver automatically repositions the arrow and the arrow's label. In contrast, a conventional language forces the designer to write code to reposition the arrow and the label. This may not seem so onerous a task, but when an application contains thousands of such relationships, the bookkeeping required to maintain them climbs so rapidly that it becomes difficult to add new functionality to the application, and the time required to debug the changes increases substantially as well.

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 need be reevaluated. An incremental constraint solving algorithm can automatically identify which constraints must be reevaluated and limit its constraint 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 allows the designer to take advantage of any special characteristics of the application to write a custom algorithm that might be faster than a general-purpose constraint solver.

Types of Constraints

Constraints can be either *one-way* or *multi-way*. One-way constraints allow the constraint solver to change only one of the objects in the constraint in order to satisfy it; multi-way constraints allow any of the objects in the constraint to be changed. For example, the arrow in figure 5 is connected to the circle and the box by one-way constraints. If either the box or circle moves, the arrow will be moved to resatisfy the constraints. However, if the arrow is moved, neither the box nor the circle will move—the constraints that tie the arrow to the box or circle will be violated or removed. If these constraints were multi-way, then the constraint solver would move the box or circle when the arrow moved, thus satisfying the constraints.

Multi-way constraints are obviously more powerful than one-way constraints, but this increased expressiveness comes at a price. One-way constraint solving algorithms are simpler to implement and more efficient than multi-way constraint solvers. This is because one-way constraint solvers only have to evaluate constraints, whereas multi-way solvers must also choose which variable in a constraint should be modified.

Multi-way constraints can also introduce ambiguity at the design level. For example, suppose we are given the constraint $A - B - C = 0$. If A changes, the constraint solver must choose whether to change B or C, or maybe to change both. In this case we would like to adhere to the *principle of least astonishment*—an editing operation by a user should change the result in a way consistent with the user's expectations[1, 2]. One approach that has been proposed for eliminating this ambiguity is constraint hierarchies[2]. This method divides constraints into hierarchies or priority levels. The constraint solver then tries to satisfy as many constraints as possible, but solves the highest priority constraints first, then the next highest priority, and so on. One issue that constraint hierarchies do not address is how to specify that multiple values should change (e.g., 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 that the user requests; eager evaluation evaluates a constraint immediately. Thus, in a lazy evaluation system, there may be 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 may 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 is interested in viewing only a limited portion of the application's data and 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 kept up-to-date), and lazy evaluation will not save many constraint reevaluations.

Constraint Systems

Examples of graphical systems that use constraint technology abound[3]. Sketchpad[4] pioneered the use of graphical constraints in a drawing editor in the early 1960s. Thinglab[1] took constraints a step further and introduced them to the realm of graphical simulation. More recently Thinglab has been refined to aid in the generation of user interfaces[2]. Both Sketchpad and Thinglab provided multi-way constraints. However, most systems that have been designed for the development of user interfaces use one-way constraints because of their simplicity and efficiency. Grow[5], Peridot[6], and Apogee[7] are three examples of such systems. Grow was perhaps the first comprehensive user interface development system that employed constraints, Peridot was the first to try to infer constraints, and Apogee was the first to employ lazy evaluation. CONSTRAINT[8] provided a user interface development environment that employed multi-way constraints, but introduced a new constraint solving algorithm that made multi-way 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 this variable are marked out-of-date. When a user requests the value of an out-of-date variable, the constraint solver demands the values of all the 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 either are atomic (i.e., not computed by a constraint) or up-to-date, at which point the constraint solver can start working its way back to the variable originally requested. In the future Garnet will probably provide a limited form of multi-way constraints and eager evaluation. We have found that the interfaces created using Garnet require almost all their constraints to be reevaluated when the display is updated; thus lazy evaluation does not avoid evaluating enough constraints to justify the increased bookkeeping that it must perform. Also, the preliminary design for the new multiway algorithm appears to be sufficiently efficient.

References

1. Alan Borning, ‘‘The Programming Language Aspects of Thinglab; a Constraint-Oriented Simulation Laboratory’’, *ACM Transactions on Programming Languages and Systems*, Vol. 3, No. 4, October 1981, pp. 353-387.

2. Bjorn N. Freeman-Benson, John Maloney, and Alan Borning, ‘‘An Incremental Constraint Solver’’, *Comm. ACM*, Vol. 33, No. 1, January 1990, pp. 54-63.

3. Alan Borning and Robert Duisberg, ‘‘Constraint-Based Tools for Building User Interfaces’’, *ACM Transactions on Graphics*, Vol. 5, No. 4, October 1986, pp. 345-374.

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

5. Paul Barth, ‘‘An Object-Oriented Approach to Graphical Interfaces’’, *ACM Transactions on Graphics*, Vol. 5, No. 2, April 1986, pp. 142-172.

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

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, October 1988, pp. 167-178.

8. Brad T. Vander Zanden, ‘‘Constraint Grammars--A New Model for Specifying Graphical Applications’’, *Human Factors in Computing Systems*, Proceedings SIGCHI’89, Austin, TX, April 1989, pp. 325-330.

Biographies

BRAD A. MYERS

Brad A. Myers is a Research Computer Scientist at Carnegie Mellon University, where he is the principal investigator for the Garnet User Interface Development Environment. From 1980 until 1983, he worked at PERQ Systems Corporation. Myers received a PhD in computer science at the University of Toronto where he developed the Peridot UIMS. He received the MS and BSc degrees from the Massachusetts Institute of Technology during which time he was a research intern at Xerox PARC. His research interests include User Interface Development Systems, user interfaces, Programming by Example, Visual Programming, interaction techniques, window management, programming environments, debugging, and graphics. He belongs to SIGGRAPH, SIGCHI, ACM, IEEE, and the IEEE Computer Society.

Address: School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3890.

DARIO A. GIUSE

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

Address: School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3890.

ROGER B. DANNENBERG

Roger B. Dannenberg is a Research Computer Scientist on the faculty of the Carnegie Mellon University School of Computer Science. He received a Ph.D. from Carnegie Mellon in 1982 after receiving a B.A. from Rice University (1977) and a M.S. from Case-Western Reserve University (1979), with fellowships from the National Science Foundation and the Hertz Foundation. 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. Dr. Dannenberg's current work in addition to Garnet includes research on music understanding, the automated accompaniment of live musicians, and the design and implementation of Arctic, a very high-level, functional language for real-time control. He also co-directs the Piano Tutor Project with the goal of applying music understanding and expert system technology to music education. Dr. Dannenberg is a member of Phi Beta Kappa, Sigma Xi, Tau Beta Pi, Phi Mu Alpha, ACM, SIGCHI, and the Computer Music Association.

Address: School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3890.

BRAD VANDER ZANDEN

Brad Vander Zanden is a post-doctoral fellow at Carnegie Mellon University, where he is working on the Garnet User Interface Development Environment. He received his bachelors degree from Ohio State, worked for a year with the firms of Arthur Andersen and Chemical Abstracts Service, and then proceeded to obtain his MS and Ph.D. from Cornell. His work at both Cornell and CMU has focused on developing new paradigms for creating graphical interfaces and on creating new incremental algorithms for constraint satisfaction. His research interests include User Interface Development Systems, user interfaces, constraint systems, incremental constraint satisfaction, programming environments, and graphics. He is a member of SIGGRAPH, SIGPLAN, ACM, IEEE, and the IEEE Computer Society.

Address: School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3890.

DAVID S. KOSBIE

David S. Kosbie is a graduate student at Carnegie Mellon University, where he is a member of the Garnet User Interface Development Environment Project. Kosbie received the BA degree from Harvard University in 1987. His research interests include User Interface Development Systems, user interfaces, and methods for simplifying graphical programming. He belongs to IEEE and the IEEE Computer Society.

Address: School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3890.

EDWARD PERVIN

Edward Pervin is a Research Programmer at Carnegie Mellon University, where he is working on the Garnet User Interface Development Environment. From 1985 to 1988, he worked at Incremental Systems Corporation, and before that at PERQ Systems Corporation. Pervin received his MS and BSc degrees in mathematics at Carnegie Mellon. Messages from /usr1/bam/mail.hg:

ANDREW MICKISH

Andrew Mickish has 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.

Address: School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3890.

PHILIPPE MARCHAL

Philippe Marchal is a former Visiting Scientist at Carnegie Mellon University School of Computer Science, where he was a Researcher in the Garnet User Interface Development Environment. A graduate from the University of Grenoble (France), he worked at DELPHIA in Grenoble, were he developed with fellows the PIXIA Prolog Interaction Toolkit before joining the Garnet team in April 1988. He is now a member of the Human Computer Interaction group of the European Computer-Industry Research Center in Munich. He is interested in user interfaces, User Interface Management Systems, interaction techniques, human factors, graphics and Object-Oriented programming.

Address: ECRC, Arabella Strasse 17 8000 Muenchen 81 Federal Republic of Germany

Addendum: Changes to Garnet Since November, 1990

Brad A. Myers

School of Computer Science
Carnegie Mellon University

Introduction

There have been significant improvements to Garnet in the three years since the accompanying article was written. This addendum highlights some of the more important changes.

The Garnet Toolkit

We have completely re-implemented the KR object system three times, and now the performance is about 10 times as fast as the original version, resulting in application performance improvements of a factor of 3 or 4.

There are also a number of improvements to the constraint system. In particular, we discovered that an important problem was the space overhead of constraints, but that up to 95% of the constraints were evaluated only once at object creation time. Therefore, we added a unique mechanism that eliminates the constraints that only depend on constant values.

The Opal object system has been enhanced with objects to support multiple-font, multiple-line text editing, including a special mode to support Lisp code editing. To make Garnet practical for use for maps and visualizations with thousands of graphical objects, we added a special feature to eliminate most of the per-object overhead when there are lots of similar objects. The multifont text object also allows arbitrary graphical objects to be embedded in the string. This new object removes the restriction that Garnet not be used for applications with significant text components.

Gesture recognition has been added to the Garnet input model so that designers can investigate advanced user interfaces. It is easy to experiment with adding gestures to existing or new direct manipulation interfaces, since gestures can be specified by example.

The widget set in Garnet has been significantly expanded. There is now a complete set of widgets with the Motif look-and-feel. These look like the standard Motif widgets, but have been implemented entirely using the Garnet low-level tools.

The debugging tools have also been substantially expanded. A new ‘‘Inspector’’ tool views and edits the properties of objects. Other views show the object hierarchies and the dependencies of constraints. We also added type-checking on the values of objects which allows programmers to catch errors earlier.

Garnet provides several editing functions in a manner that should be usable by most graphical editors *without change*, unlike other application frameworks where programmers must subclass these operations. These include cut, copy, paste, delete, duplicate, group, ungroup, refresh, to-top, to-bottom, save, open, etc. Garnet will also create a postscript file for any window, so printing is easy.

Higher Level Tools

There are two entirely new tools in Garnet. The first is the Gilt interface builder, which allows users to select widgets from a palette and place them into a window. Gilt provides access to the full set of widgets, including pop-up widgets like menubars and option buttons. When complete, the interface design can be tested and saved to a file. There are two important innovations in Gilt: dependencies among widgets and filtering of values can be demonstrated[1], and graphical styles can be defined so that widgets will look consistent across multiple dialog boxes and applications[2].

The Marquise tool[3] allows the overall interactive behaviors of an application to be described by demonstration. Other Garnet tools allow pieces of an application to be created, but a tool to put it all together had been missing. With Marquise, the user can demonstrate the overall design of application windows and how the various widgets control the behaviors. Innovative features in Marquise include: the ability to demonstrate graphical palettes that control what kinds of objects are created and the properties of objects (color, font, etc.); control over where the user must click for a behavior to start, and special icons to show where the events occurred.

Distribution

Garnet is now in the public domain, and it is available by anonymous FTP. To get Garnet, FTP to `a.gp.cs.cmu.edu`. Change to `/usr/garnet/garnet/` and retrieve `README` for instructions. Or you can send electronic mail to `garnet@cs.cmu.edu`. The distribution includes the entire toolkit, plus Gilt, C32 and Lapidary. We make sure that Garnet works in any Common Lisp environment, including Allegro, Lucid, CMU, Harlequin, AKCL, and CLISP Common Lisps on Sun, DEC, HP, Apollo, IBM 6000, SGI, and many other machines. As a result, there are now over 50 projects all over the world using Garnet to create a wide variety of applications.

Acknowledgements

Additional members of the Garnet project since 1990 include: James Landay, Rich McDaniel, Matt Goldberg, Rajan Parthasarathy, and Don Hopkins.

References

1. 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, November 1991, pp. 211-220.

2. Osamu Hashimoto and Brad A. Myers, ‘‘Graphical Styles for Building User Interfaces By Demonstration’’, *ACM SIGGRAPH Symposium on User Interface Software and Technology*, Proceedings UIST’92, Monterey, CA, November 1992, pp. 117-124.

3. 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.