DECLARATIVE PROGRAMMING IN A PROTOTYPE-INSTANCE SYSTEM: OBJECT-ORIENTED PROGRAMMING WITHOUT WRITING METHODS

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

Dario A. Giuse
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
dzg@cs.cmu.edu

Brad Vander Zanden
Computer Science Department
University of Tennessee
107 Ayres Hall
Knoxville, TN 37996-1301
bvz@cs.utk.edu

ABSTRACT

Most programming in the Garnet system uses a declarative style that eliminates the need to write new methods. One implication is that the interface to objects is typically through their data values. This contrasts significantly with other object systems where writing methods is the central mechanism of programming. Four features are combined in a unique way in Garnet to make this possible: the use of a prototype-instance object system with structural inheritance, a retained-object model where most objects persist, the use of constraints to tie the objects together, and a new input model that makes writing event handlers unnecessary. The result is that code is easier to write for programmers, and also easier for tools, such as interactive, direct manipulation interface builders, to generate.

KEYWORDS: Object-Oriented Programming, Prototype-Instance Model, Toolkits, Declarative Programming, Constraints, Input, Garnet.

1. Introduction

Over the last three years of using the Garnet system to create dozens of large-scale user interfaces, we have observed that the style of programming in Garnet is quite different from that in conventional object-oriented systems. In Garnet, programmers combine pre-defined objects into collections, use constraints to define the relationships among them, and then attach pre-defined ‘‘Interactor’’ objects to cause the objects to respond to input. The result is a declarative style of programming where the programmer rarely writes methods. Furthermore, the interface to objects is usually through direct accessing and setting of data values, rather than through methods.

The features of the Garnet object system have been motivated by the overall goal of the project: to provide high-level, interactive, mouse-based tools for rapidly prototyping and creating graphical, highly-interactive, direct manipulation programs. Because of the emphasis on rapid creation and easy editing, we have chosen to make the object system completely flexible and dynamic. Since the interactive tools need to be able to generate code for the interface and then read the code for later editing, it is easier to generate high-level, declarative specifications. Because much of the look and the dynamic behavior in Garnet can be specified by supplying parameters to pre-defined objects, it is easier for interactive tools to display these options in dialog boxes or intelligently guess them using demonstrational techniques.

In order to achieve these goals, we have made a number of interesting design decisions which contribute to Garnet's unique programming style. First, Garnet uses a prototype-instance model rather than the more popular class-instance model. In a prototype-instance model, there is no distinction between instances and classes; any instance can serve as a ''prototype'' for other instances. Garnet's model is unique in that it supports structural-inheritance. This means that when a prototype object is a collection (or ''aggregate'') of other objects, Garnet creates instances of all components when the aggregate is instanced. Therefore, the programmer can construct complex graphical objects by declaratively listing the primitive component objects. It is not necessary to write creation or drawing methods.

Second, the objects in Garnet are usually persistent and long-term. For example, the graphics model requires that there be an object in memory corresponding to each object on the screen. This means that the programmer does not have to deal with object refresh, and allows the toolkit to contain high-level support, like selection handles. In many other systems, a single object can be used like a stamp-pad and drawn in multiple places on the screen.

Third, constraints can be used to declare the relationships among the objects. Constraints in Garnet are tightly integrated with the object system, so that any slot of any object can have a constraint which calculates its value. The result is that the interface to objects is usually through data values which are directly accessed and set, rather than through methods. Constraints are used to propagate the changes appropriately.

Fourth, Garnet incorporates a novel input model, which provides standard objects called ''Interactors'' to handle the most popular direct manipulation behaviors. This is based on the Model-View-Controller idea from Smalltalk [8], where the Interactors correspond to the controllers. In Garnet, however, unlike in Smalltalk and other implementations of this idea, the programmer rarely writes new Interactor methods. Instead, the programmer attaches an instance of a pre-existing Interactor object to the graphical objects using constraints, and declaratively specifies any necessary controlling parameters for the Interactor.

This paper discusses these aspects of Garnet, and shows the advantages of the Garnet style of programming. Even though conventional wisdom for object-oriented programming is that writing methods is ''good'' and exposing the objects' data is ''bad,'' we show that the Garnet style is just as modular and provides just as much information hiding. Furthermore, there is some evidence that, at least for user interface programming, it is more effective.

Garnet is a comprehensive user interface development environment in Lisp for X/11.[1] It is in the public domain and is freely available. Currently, over 30 projects around the world are using the system regularly.[2] The system contains a number of features that make it well-suited for creating graphical user interfaces. Unlike other toolkits which primarily supply widgets, Garnet is specifically designed to cover *all* aspects of user interface programming, especially the insides of application windows. While there have been a number of papers about Garnet [17] and its components [23, 15, 24, 19], this is the first paper about the programming *style*. For a complete discussion of programming in Garnet, see the reference manual [20].

2. Related Work

In the terms of the ''Treaty of Orlando'' [22], the Garnet object system is a prototype-instance model with dynamic, implicit, per-object sharing. It is dynamic because the inheritance can be changed at any time, implicit because objects inherit from their prototypes and you cannot explicitly declare how slots are inherited (except by using constraints), and per-object because there is no such thing as classes. The ''templates'' (prototypes) are entirely ''non-strict,'' which means that an instance can gain or lose slots at any time. These features make Garnet much like SELF [5] and other prototype-instance systems [9]. However, unlike SELF, Garnet rarely uses multiple inheritance (although it is allowed), and we have integrated a constraint solving mechanism with the

[1]Display Postscript and Macintosh versions are in progress.

[2]You can get Garnet by anonymous FTP from `a.gp.cs.cmu.edu`. Change to the directory `/usr/garnet/garnet/` (note the double `garnet`'s) and retrieve `README` for instructions. Or you can send electronic mail to `garnet@cs.cmu.edu`.

object system. Another important difference is that Garnet encourages programmers to directly access and set slots of objects, whereas SELF prevents this and only provides methods.

Many systems have used constraints as part of an object system [3], but none is as general-purpose or fully-integrated as Garnet. Garnet is also the first system to introduce pointer variables into constraints (where the objects referenced by the constraint can change). The first integrated constraint and object system was ThingLab [2], which supported multi-way constraints. ThingLab was also a prototype-instance object system. Apogee [7] and Grow [1] are more closely related to Garnet in goals, since they are user interface toolkits. Also, like Garnet, they implement one-way constraints. Neither, however, uses constraints as the primary mechanism for information passing, so they both make extensive use of methods.

As was mentioned, Garnet's input model is based on the Model-View-Controller idea from Smalltalk [8]. Other attempts to capture interactive behaviors include the model used by graphics standards, such as PHIGS, GKS, CGI, CORE, etc., which identifies five or six basic input types (e.g., locator, stroke, valuator, choice, pick and string for PHIGS). This is based on a model by Foley and Wallace [6]. Unfortunately, this model has proven unusable for modern user interfaces [12].

The current object and constraint system is a complete redesign and rewrite of the Coral system [23]. Coral was implemented in CLOS, but was abandoned because it was too slow and inflexible in practice. Like Garnet, Coral provided a declarative syntax for objects and constraints, but it was not possible to modify objects once they had been created. Coral used a conventional class-instance model, rather than the prototype-instance model we now use. It also required that the constraints be parsed to search for object references, which limited the kinds of constraints that could be written. The current Garnet constraint system does not need to parse constraints because it dynamically determines the dependencies when the constraint is evaluated. Coral did not provide for arbitrary pointer variables in constraints like Garnet does now, and it used active values, which we have found to be unnecessary in Garnet with fully functional constraints. Coral had a special-purpose mechanism for constraints over lists of objects, such as the items of a menu. For example, you could specify a constraint for the top of the first item and a different constraint for the rest of the items. This is not needed in Garnet due to the support for arbitrary code in constraints (you can just use Lisp's looping facilities). The create routines in Coral were specific to each class, rather than a generic function that would work for all classes. Other important problems with Coral were that the declarative technique did not support changing objects after they were created, and it was not available to interactive editors. Therefore, a separate procedural mechanism was supplied. In the current Garnet, the declarative and procedural mechanisms have equivalent power.

3. The Prototype-Instance Object Model

The Garnet object system implements the prototype-instance model [9], and supports completely dynamic redefinition of prototypes with automatic change propagation. 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''). An instance can add any number of new slots, and slots that are not overridden in an instance inherit the values from its prototype. In fact, the inheritance can change dynamically, as an object can add or remove slots at any time. There is no distinction 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. This allows the methods that implement messages to change dynamically, which is not possible in conventional object systems like Smalltalk. The ability to dynamically add, delete, and modify methods has proven important in graphical interface builders since they need to temporarily insert their own methods during ''build'' mode, and then retract them during ''test'' mode.

All objects are created with the standard function `create-instance` which takes an optional name of the new object, an optional object to be used as a prototype, and a list of slots and values that should have local values. Slots that are not mentioned start out using the inherited, default value from the prototype (which can be changed later). Slot names start with colons, and can contain any number of printable characters (e.g., `:left`, `:interim-selected`, `:obj-over`). In the following example, the rectangle named `my-rect` will inherit the `:top`, `:width`, and `:height` from the prototype `rectangle`:

```
;create an object named rectangle inheriting from nothing.
(create-instance 'rectangle NIL
    (:top 10) (:left 10)  ;specify values for some slots.
    (:width 20) (:height 25) (:color black))

;create my-rect inheriting from rectangle.
(create-instance 'my-rect rectangle
    (:left 45)(:color blue)) ;override two slots.
```

Setting an object's slot with a value automatically creates the slot, if needed. This makes it extremely easy to associate any piece of information with any object, since slot names do not have to be predefined. For example, the following will create a new slot in `rectangle`:

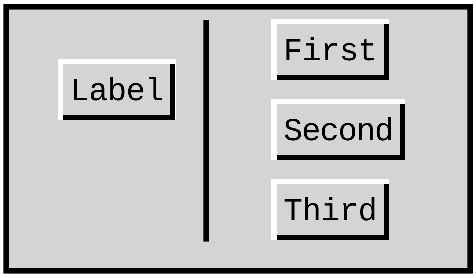
```
(s-value rectangle :perimeter 90)
```

Because `my-rect` inherits from `rectangle`, it will also now have the new slot.

There is a special kind of object in Garnet called an ''aggregate'' which is a collection of other objects. A unique feature of Garnet is that whenever an instance is made of an aggregate, Garnet automatically creates instances of all its components, and links them together appropriately. This ''structural inheritance'' is an extremely powerful abstraction, because it frees the user from having to know whether an object being instanced is a primitive object like `rectangle` or a composite like `button`; the create-instance call is the same.

For example, a button might be composed of three rectangles and a text object. The programmer can declaratively list these as part of a button, as shown in Figure 1. Then, when the user creates `my-button1` using `button` as the prototype, Garnet automatically creates instances of the three rectangles and the text. Of course, any of the parts could themselves be aggregates, and the instancing would be applied recursively. Constraints (described below) are used to declare how the properties of the components are connected.

An important innovation in Garnet is that edits made to the prototype are automatically reflected in all instances. For example, if the color of `fill-inside`



(a)

```
(create-instance 'button aggregate
  (:parts
    ((:top-edge rectangle ....)    ;white left & top edges
     (:bottom-edge rectangle ....);black right & bottom
     (:fill-inside rectangle ...) ;grey interior
     (:label text ...             ;string inside button
             (:string "Label")))))

(create-instance 'my-button1 button
    (:left 100)(:top 5)(:string "First"))
(create-instance 'my-button2 button
    (:left 100)(:top 35)(:string "Second"))
(create-instance 'my-button3 button
    (:left 100)(:top 65)(:string "Third"))
```

(b)

Figure 1:
(a) A button (shown on the left) and some instances created from it. (b) The outline of the button's aggregate structure and the code to create the instances.

were changed in `button`, it would automatically also change in `my-button1` and all the other instances (see Figure 2). More significantly, if a part is added or removed from the prototype, then Garnet will add or remove the corresponding object from all instances. For example if `top-edge` was removed from `button`, then the appropriate rectangle would also be removed from `my-button1` and the other instances. Garnet stores pointers in each prototype to all instances to support these operations.

Similarly, if the programmer wants to create an object which is a slight modification of an existing object, it is only necessary to override the divergent parts. For example, the programmer could have left the existing `button` prototype of Figure 1 unmodified, and created a new type of button that looks like Figure 2 by specifying:

```
(create-instance 'new-button button
  (:parts
   ((:top-edge :omit) ;don't want the top-edge rectangle.
    (:fill-inside :modify
          ;just change the filling-style property.
          (:filling-style light-gray))
    ;;  :bottom-edge and :label are unchanged.
    )))
```

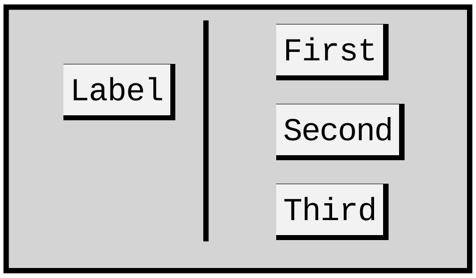


Figure 2:
When the color of the `fill-inside` rectangle is changed to light-gray and the `top-edge` rectangle is removed from the button prototype in Figure 1, these changes propagate automatically to the instances.

In a conventional object system, the programmer would instead be required to rewrite the entire `draw` method (and probably the `erase` and many other methods as well). In Garnet, only the specific parts to be changed need to be mentioned, and only in the object definition.

A very significant advantage of this technique is that it is possible to provide graphical, interactive tools that will create the graphical objects. For example, Lapidary [15] allows programmers to draw pictures of new widgets (like the buttons above) and of new application-specific prototypes. The interface of Lapidary is much like a conventional drawing program like MacDraw. The programmer can specify which slots will be parameters (for the button, they might be the position and string). Interactive behaviors and relationships among the components can also be defined. Because all objects have the same structure, Garnet provides a built-in routine that will save the objects to a file [21]. The contents of the file is simply the declarative code to create the objects, as in Figure 1-b. Therefore, this file can be compiled using the standard Lisp compiler, and the standard Lisp `load` routine is all that is needed to read in the objects.

Therefore, Lapidary can simply call the standard save routine to write the created objects to a file, instead of having to generate code for the methods to create, draw, and erase the objects, and for handling input events. When the application wants these graphical objects to appear at run-time, it only needs to load the file, and create instances of the prototypes supplying the appropriate parameters. Note that unlike other interactive interface builders, such as the NeXT Interface Builder, Lapidary allows the designer to define *entirely new* objects, not just choose pre-defined objects from a palette. The various features of Garnet's object system make this much easier to implement [21].

Since edits to a prototype are reflected in its instances, it is even possible to interactively change the appearance of objects *while* they are being used in an application. When Lapidary or a similar tool changes the prototype, all of the instances are updated immediately, even if they appear inside of an application that is currently running. This helps achieve the goal of making Garnet useful for rapid prototyping of interfaces, since the designer can see the results of the edits in context. In a class-instance model or any method-based object system, it would usually be necessary to stop and recompile to see the results of edits.

One claimed disadvantage of the prototype-instance model is speed, since every slot access and setting might require a search up the inheritance hierarchy to find the slot. However, through implementation techniques such as caching, we have significantly improved the performance of Garnet. Thus, even though Garnet offers dynamic inheritance, constraints, and automatic constraint elimination (explained below), it only takes 17.9 microseconds to access a slot (on a SPARCStation 1, using Allegro Common Lisp v4.0.1).

4. Retained Objects

Another important feature of Garnet's object system is that most objects are ‘‘long-term.’’ Unlike other object systems, it is rare in Garnet to repeatedly allocate and dispose of objects. Most objects are used to represent application information, graphical displays, or interactive behaviors which persist.

For example, all graphics use a ‘‘retained-object model’’ (sometimes called ‘‘structured graphics’’ or a ‘‘display list’’). This means that for every graphical object on the screen, there is a corresponding object in memory. Therefore, to make something appear on the screen, the programmer creates instances of graphical objects and adds them to a window. A significant difference from other systems that supply structured graphics, such as CLIM [11] and InterViews [10] is that there is no way to avoid using the structured graphics in Garnet: *all* graphics must be displayed by attaching instances of objects to windows.

As an example, to display `my-button1` or `my-rect`, the programmer can create an instance of a window and add these objects:

```
(create-instance 'my-window window)
(add-components my-window my-button1 my-rect)
```

This will cause the objects to be displayed. Prototypes can also be displayed, since there is no distinction between prototypes and instances. Therefore, the button that says ''Label'' in Figure 1 is the actual prototype for the instances.

In a similar way, Interactor objects which control behaviors (described below) are also allocated and attached to graphics. Furthermore, the data that describe the information and state of the application are often stored as Garnet objects. Thus, our techniques are not just limited to the graphical user interface part of the application.

In order to change any property of an object, it is only necessary to set the appropriate slot, and Garnet will propagate the change appropriately. For example, to change the string of `my-button1`, you could use:

```
(s-value my-button1 :string "New Label")
```

This is implemented using a special demon procedure that can be associated with each object. This demon will be called whenever any slots of the object change. For graphical objects in Garnet, a built-in demon is used which automatically insures that the appropriate graphical objects on the screen are redrawn. The graphical update algorithm attempts to minimize the number of objects that are redrawn by first determining all objects that change and all objects that intersect them, and then drawing only those objects (from back to front) using an appropriate clipping region. A different demon is used for Interactor objects, and applications can supply their own demons for application-specific objects, if necessary.

The advantage of the retained-object model is that programmers are freed from many of the maintenance tasks they would have in most other systems. There is never a need to write or call `create`, `initialize`, `draw`, or `erase` methods. When a complex application-specific graphical object is desired, the programmer uses the declarative syntax to list all the component parts, and then creates instances and adds them to the appropriate window. Of course, the prototypes themselves can also be created dynamically at run time. When objects are to be changed, Garnet automatically determines what must be redrawn. Using the same mechanism, Garnet handles window scrolling and refresh automatically.

Of course, the primitive graphical objects, such as rectangles, lines and text, use draw methods internally to display themselves on the screen. Other internal methods are used for handling refresh and for asking objects whether they are under the mouse. However, since Garnet supplies a primitive object for each kind of drawing operation in the X Window System, anything that can be drawn in X can be created by combining instances of Garnet's graphical objects. Therefore, the programmer can simply combine the built-in graphical objects, and never needs to write new methods.

Another important advantage of the retained model is that the toolkit can provide built-in utilities for many of the common functions, since all data uses a standard structure. For example, Garnet provides a widget which displays the popular square ''handles'' around graphical objects for selection, moving, and growing them. This works because the handles can reference the retained graphical objects to know what is on the screen, and how to modify them. Similarly, there are built-in routines for creating, duplicating, deleting, moving, growing, and printing objects. Thus, application developers do not need to write code for any of this.

The primary problem with the retained object model is the potential for enormous space inefficiencies. If there are 10,000 objects on the screen, there must be 10,000 objects in memory to represent them. We have taken a number of steps to overcome this problem. As with the Glyphs in InterViews [4], we remove unneeded information from objects. For example, we can remove large numbers of unnecessary constraints (see below). However, unlike Glyphs, each object in Garnet still keeps information about where it is located on the screen. Second, if there are a large number of nearly identical objects, such as the squares in a bitmap editor (‘‘fat bits’’), the lines in a map or mesh (Figure 3), or the dots in a graph, then a ‘‘virtual aggregate’’ can be used that just pretends to create an object for each graphic. The programmer provides a prototype object, and the virtual aggregate simulates creating an instance for each data value, but actually does not allocate any objects in memory. It still appears to the rest of the code, however, that there is an object for each value. Using these techniques, people have created quite large applications using Garnet.

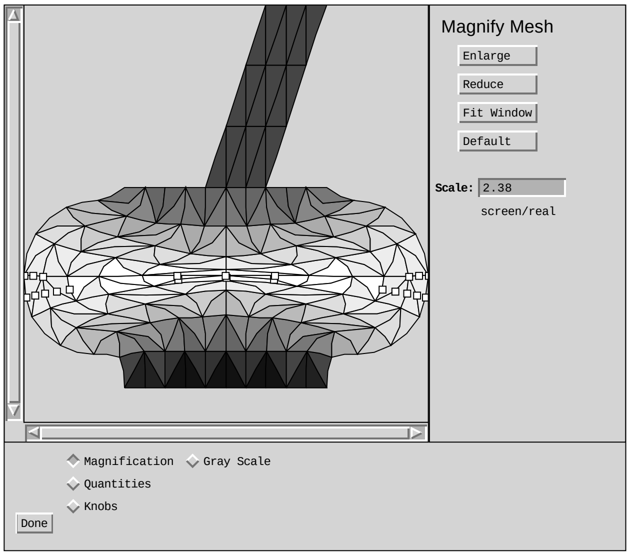


Figure 3:
A mesh created using a virtual aggregate for the polygons and another virtual aggregate for the square knobs. For the polygons, the virtual aggregate is passed a prototype for a polygon, and an array containing the list of points and the color for each polygon. The virtual aggregate then pretends to allocate an object for each element of the array, but actually just draws the prototype object repeatedly. (Picture courtesy of Kenneth Meltsner of General Electric [13].)

5. Constraints

An important feature of Garnet is that any slot of any object can contain a *constraint* instead of a normal value. A constraint is a relationship that is declared once and then maintained automatically by the system. For example, instead of making one endpoint of a line be (10,45), a programmer can define it to be the same as the center of the left edge of a rectangle. Then the system will change the value of the endpoint automatically whenever the rectangle moves. The syntax for referencing slots of objects in Garnet is `(gv object slot)`, where `gv` stands for ‘‘get-value.’’

Although many other research systems have provided constraints, Garnet is the first to truly integrate them with the object system and to make them general purpose. Constraints in Garnet can be any Lisp expression. An important result of these design decisions is that constraints are used throughout the system in many different ways. For example, Garnet’s implementation of a Motif radio button widget uses 58 constraints internally, and the Lapidary graphical editor, which is a large and complex application, contains 16,700 constraints. Of course, many of these are only evaluated once, and may be eliminated, as will be discussed later.

Since they can contain arbitrary code, constraints might be thought to be like methods, and, in fact, they serve a similar purpose: to define the operation of objects. However, the important point is that programming with constraints is a different style than programming with methods, in the same way that programming with methods is a different style than conventional procedural programming. For one thing, constraints are automatically evaluated when necessary, rather than requiring the programmer to invoke them at appropriate times. Secondly, constraints are declarative, in that they compute the values of variables (slots) based on values of other variables, and do not have side effects. Finally, by focusing on data values, constraints make programming more data oriented, rather than procedure oriented. Section 8 discusses why constraints provide more information hiding than conventional methods.

One obvious use of constraints is to tie parts of composite objects together. When the programmer collects together a set of objects to make a composite, it is necessary to specify how the parts relate. Garnet provides a declarative syntax so the programmer can simply list the relationships of the parts. An innovation of the Garnet constraint system is that the objects can be referenced through pointer variables [25]. This is used to allow the code of the constraint to be independent of the specific objects used for the parts. Instead, the constraint will reference the object using a ‘‘path’’ through the aggregate hierarchy. For example, in the button of Figure 1, the `bottom-edge` rectangle can refer to the width of the text object using:

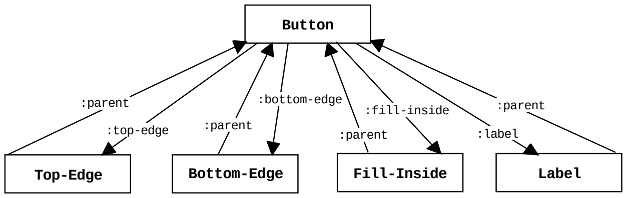
```
(gv :SELF :parent :label :width)
```

As shown in Figure 4-a, this starts from the `bottom-edge` rectangle, goes up to the parent aggregate, down to the `label` part, and gets the `:width` from there. Thus, the width of the `bottom-edge` will be the same as the width of the `label`. This will work in the prototype, as well as in all instances, since Garnet sets pointers to the appropriate objects into the slots `:parent`, `:fill-inside`, `:bottom-edge`, etc. This makes it easy for Garnet to create instances of the entire aggregate (including the constraints), since Garnet does not need to edit the constraint code. Because this style of constraint is quite common, we provide an abbreviation of `(gv :SELF)` as `gvl`. Figure 4-b shows the constraints used to tie together the parts of the button of Figure 1.

These constraints are fairly simple, and are representative of the majority of the constraints used in Garnet. However, some objects have quite long and complex constraints. For example, the `aggregraph` object is a special type of aggregate that displays its components as a tree or graph, and it has a very large constraint that computes the graph layout information.

Another important use of constraints is to copy values and parameters around. For example, the Motif button prototype takes the string label, the color, and the position as parameters (among others). These parameters are supplied as values in the slots of the top-level widget aggregate. When the object is created, the programmer can specify whichever slots need different values and the rest are inherited. Of course, any value can be changed later while the widget is displayed, if desired. Note that this is quite different from a conventional system that requires the widget creation method to take a large parameter list with all possible values to be set, and therefore requires a custom creation method for each object. In Garnet, the standard `create-instance` routine is used for all objects, and it can be used to set an arbitrary number of slots.

Although the slots which serve as parameters are in the top-level button aggregate, for these values to actually take effect they must be copied down to the appropriate places in the components. For example, the string value is specified at the top level in Figures 1 and 4-b, but it is needed by the text object. So there is a constraint in the text object that copies the value of the parameter. Of course, since constraints can be arbitrary Lisp code, the values can be transformed arbitrarily as needed. Since constraints are used to



(a)

```
(create-instance 'button aggregate
 (:left 20)          ; These are the
 (:top 20)           ;  parameters to
 (:string "label") ;  the button.
 (:parts
  ((:top-edge rectangle
     (:left (formula (gvl :parent :left)))
     (:top (formula (gvl :parent :top)))
     (:width (formula
        (+ (gvl :parent :label :width) 8)))
     (:height (formula
        (+ (gvl :parent :label :height) 8)))
     (:color white))
   (:bottom-edge rectangle
     (:left (formula (+ 2 (gvl :parent :left))))
     (:top (formula (+ 2 (gvl :parent :top))))
     (:width (formula
        (+ 6 (gvl :parent :label :width))))
     (:height (formula
        (+ 6 (gvl :parent :label :height))))
     (:color black))
   (:fill-inside rectangle
     (:left (formula
        (gvl :parent :bottom-edge :left)))
     (:top (formula
        (gvl :parent :bottom-edge :top)))
     (:width (formula
        (- (gvl :parent :bottom-edge :width) 2)))
     (:height (formula
        (- (gvl :parent :bottom-edge :height) 2)))
     (:color gray))
   (:label text
     (:left (formula
        (center-x (gvl :parent :fill-inside))))
     (:top (formula
        (center-y (gvl :parent :fill-inside))))
     (:string (formula (gvl :parent :string)))))))
```

(b)

Figure 4:
(a) The structure of the objects in the button of Figure 1 showing the references. (b) The complete code used to produce the button. This shows the constraints which put the graphics in the correct places and copy the parameter values to the parts.

propagate the values, the objects do not have to do anything special to allow changes at run-time: if one of the parameter slots is changed, the constraints automatically propagate the change appropriately, and the update algorithm will make sure the object is then redrawn.

An interesting observation about this use of constraints is that it allows *arbitrary* delegation of values, not just from prototypes. Any slot can get its value from any slot of any other object through constraints. Therefore, the constraints can be used as a form of inheritance. Of course, constraints are more powerful than conventional inheritance since they can perform arbitrary transformations on the values.

As with the graphical objects themselves, constraints can be defined interactively using various editors. Lapidary provides some iconic menus for defining the most popular constraints (Figure 5). We have found that these are sufficient for most graphical applications. For more complex constraints, a spreadsheet-like interface, which is called C32, provides a number of features to help programmers who do not know the exact syntax [19]. For example, C32 has menus that will insert commonly used functions. Also, the user can point to objects with the mouse and C32 will insert a reference into the constraint using the correct path expression. Of course, it also balances parentheses. In the future, we will explore automatic inferencing of constraints, as was done in Peridot [14]. We envision that when ‘‘guessing’’ mode is turned on, the system will try to find a likely constraint between the newly drawn object and the neighboring objects.

The performance of constraints in Garnet is quite fast. Evaluating constraints is not much slower than the calculations the programmer would have to perform anyway. On a Sun SPARCstation 1, a simple constraint evaluation (in Lisp) takes 110 microseconds. This means that objects tracking the mouse can afford to have dozens of constraints being re-evaluated for each incremental mouse movement. The system caches old values for constraints, so ones that do not change value are not re-evaluated. We have discovered that the primary performance problem with constraints is not speed, but rather space. For each constraint there must be pointers from slots that are

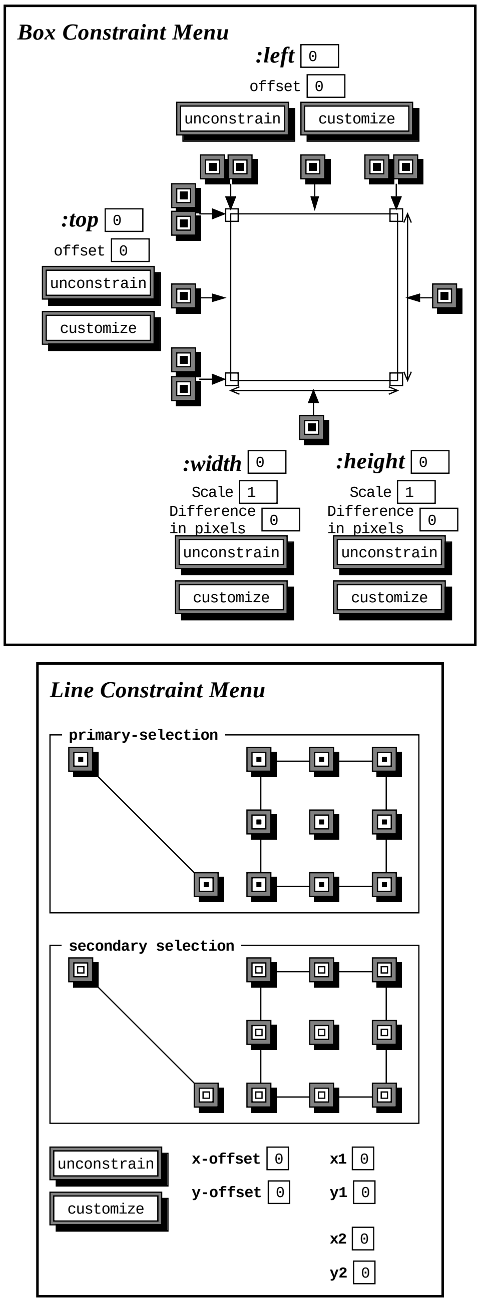


Figure 5:
The dialog boxes from Lapidary [15] that allow the most common constraints to be set. The menu on the top is for rectangular objects (which includes circles and aggregates), and the one on the bottom is for attaching lines to each other or to rectangular objects. For the box constraints, the column of buttons labeled ***:top*** will cause the dependent object to be: on top of the other object, just inside the other object, centered vertically in the object, just above the other object, or just below the other object. Similarly, the row of buttons labeled ***:left*** determine the horizontal relationship. The button on the bottom constrains the width, and the one on the right constrains the height. The text fields, like `offset` and `scale` are used to supply parameters to the constraints. For lines, either end of the line can be attached to various positions of a box-like object or of another line.

referenced to the constraints that use them, and from constraints to the slots they reference. We observed that many of the constraints are only used once when the object is initially placed, so we devised a technique where no memory is allocated for these constraints. This has been enormously effective, and decreases the total run-time storage requirements of applications on average by about 50%. For some dialog boxes, like the color selection palette, 1500 constraints are reduced to only 100. As an example of a large scale application, 6690 constraints (which is over 40%) are eliminated from the Lapidary graphical editor.

The use of constraints provides the programmer with a number of important benefits. The most obvious is that the system maintains the relationships among objects that otherwise would be the responsibility of the programmer. More relevant to this paper, however, is that constraints allow objects to provide an abstract interface through top-level variables, and the programmer can declaratively specify how to transform the values for all components. In fact, if you need to use methods, constraints can even be used to dynamically determine which method to use for a message based on the current state. This works because the value of any slot can be computed using a constraint, and the value returned can be a function. However, we do not know of anyone using this feature.

6. Input Model

Virtually all toolkits, graphics packages, and window managers use the same input model: a stream of input event records is sent to the appropriate window. The application program is expected to de-queue these events and interpret them. Garnet uses an entirely different model, based on encapsulating input behaviors separately from the graphics [16, 18]. This handles all input so objects never need event-handling methods.

Garnet provides seven basic ‘‘Interactor’’ objects that handle all of the most common direct manipulation behaviors. The Interactor objects in Garnet are completely independent of any graphical representation, and are purely input filters.[3] The seven types of Interactors currently in Garnet are:

`Menu-Interactor` - Used to select one or more from a set of objects. This can be used for menus, radio buttons, check boxes, simple push buttons, and the arrows on scroll bars. In addition, this can be used to cause application objects to become selected in a graphics editor.

`Move-Grow-Interactor` - This is used to move an object or one of a set of objects using the mouse. There may be feedback to show where the object will be moved, or the object itself may move with the mouse. This Interactor can be used to implement the indicator for one-dimensional or two-dimensional scroll bars, and also for moving application objects in a graphics editor.

`New-Point-Interactor` - This is used when one, two or an arbitrary number of new points are desired from the mouse.

`Angle-Interactor` - This is used to get the angle the mouse moves around some point. It can be used for circular gauges or for rotating objects.

`Trace-Interactor` - This is used to get all of the points the mouse goes through between start and end events, for use in free-hand drawing.

`Text-String-Interactor` - This is used to edit text and supports single-line, or multi-line and multi-font strings. A key translation table allows arbitrary mappings of editing operations.

`Gesture-Interactor` - This supports freehand gesturing, like drawing an "X" on top of an object to delete it.

Unlike other implementations of the Model-View-Controller idea, in Garnet the programmer never needs to create new kinds of ‘‘controllers.’’ It is only necessary to create an instance of a pre-defined Interactor and to supply a few parameters. An important reason that this works is that we have carefully chosen the parameters so that they support the full range of direct manipulation interfaces. For example, the designer can specify which mouse button or keyboard key causes the Interactor to start operating, and which causes it to stop. Menu-interactors can be told whether single or multiple selections are desired. The most important parameters, however, are the

[3] Note that this use of the term ‘‘Interactor’’ is different from some other systems that use the term for an entire widget (graphics plus behaviors). In Garnet, Interactors have no graphics, only behavior.

graphics that the Interactors operate over. We have observed that although direct manipulation interfaces vary widely in their ‘‘look,’’ they are mostly identical in their ‘‘feel’’ or behavior. Therefore, by separating the behavior from the graphics, and including parameters for the most popular options, virtually all behaviors can be provided without requiring new code.

For example, to create an interactor which moves around any of the objects which are components of an aggregate called `my-agg`, the following is all that is needed:

```
(create-instance 'my-mover Move-Grow-Interactor
   (:feedback-obj my-feedback-rect)
   (:start-where '(:element-of my-agg)))
```

The rest of the properties of `my-mover` will use the default values (start on left button down, move the object rather than grow it, etc.). After it is created, `my-mover` will continuously watch for a mouse leftbutton press over any of the objects in `my-agg`. When this happens, it will make the feedback object (`my-feedback-rect`) visible and begin moving it to follow the mouse until the mouse button is released. At that point, the `my-feedback-rect` will become invisible and the actual object will be moved. (If no feedback object had been supplied, then the element of `my-agg` would be directly dragged by the mouse.)

There is a standard protocol through which the Interactors interface to the graphical objects. The `move-grow-interactor` sets the `:box` slot of objects, and the `:left` and `:top` slots would be tied to the `:box` slot with constraints. This allows there to be arbitrary filtering without the Interactor having to know about it. To find which object is under the mouse, the Interactor sends a message to the aggregate. This will, in turn, send messages to each of the components. However, the programmer never has to write methods for these, since all graphical objects are created by combining the Garnet primitives which supply the appropriate methods.

The `Menu-Interactor` has two protocols: it can take a separate feedback object as a parameter, or it will directly modify the object that becomes selected. If there is a feedback object, then its `:obj-over` slot is set to the object that becomes selected. The feedback object is expected to have constraints that will cause the position and size to depend on whatever object is set into the `:obj-over` slot. For example, the left formula might be `(gvl :obj-over :left)`, which will make the feedback object have the same left position as whatever object is selected. Notice that the Interactor does not need to know whether the feedback object is a simple XOR rectangle or an aggregate containing squares that serve as selection handles.

If there is no feedback object, then the `menu-interactor` sets the `:selected` slot of the object itself. There might be constraints that change position, color or font based on whether the object is selected or not. For example, to implement a Motif-like pushed-in appearance for the button of Figure 6, the color of the `:top-edge` might be computed by the constraint:

```
(if (gvl :parent :selected)
    black ; then case
    white) ; else case
```

The formula on the `:bottom-edge` would be the opposite, and the color of the `fill-inside` would choose between gray and dark-gray. Note that this is all performed without methods: the parameters to the Interactors are values in slots, and the interface between the Interactors and the graphical objects is through setting well-defined slots in the graphics.

It is always legal in Garnet to set a slot’s value (the slot does not have to be pre-defined). Therefore, if the programmer does not want anything to happen when the object becomes selected, he or she can simply not attach any constraints to the slots. There is never a worry of a ‘‘Message-not-understood’’ error as in a conventional class-instance system, where the programmer would have to define an appropriate method at the root class (e.g., `object`), to make sure that there would never be a run-time error if arbitrary objects could be selected.

Since Interactors can be specified by filling in parameters, it is easy to create them in interactive editors. For example, Lapidary provides a dialog box for each Interactor type that allows graphics to be attached and parameters to be set. This is how Lapidary allows arbitrary behaviors to be connected to application-specific graphics interactively, without requiring the programmer to write code. Interactors can be added to aggregates, so the single

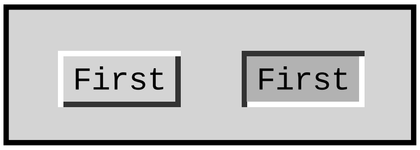


Figure 6:
The button of Figure 1 can be made to look like it moves in 3-D by changing the colors of the parts. The Interactor does not need to know how the button responds to becoming selected.

`create-instance` call will create the graphics and Interactors necessary for an object to behave correctly.

We have found the Interactor model to be extremely effective. This model makes it much easier to program direct manipulation interfaces. However, we have found a few cases where the built-in parameters are not sufficient. In this case, it is possible for the programmer to write methods to filter the data. Typically, these are used when custom processing is needed when the Interactor starts, stops or aborts. Even when this is required, however, the interface the programmer sees is still higher-level than conventional event-handling. Details are available elsewhere [20].

7. Example and Comparisons

To give an example of the style of programming in Garnet, we will sketch the implementation of the toy graphics editor in Figure 7 and compare it with the implementation in conventional object-oriented languages. Here, every time the user clicks with the right mouse button in the drawing window, a new box and arrow is created using the current line style (which is shown on the left). The arrows always go to the previously-created box. The user can click with the left mouse button to select objects, and the handles appear. Dragging a handle moves or grows the selected object. The `Delete` button deletes the selected object, and pressing on a new line style while an object is selected causes the object to change. Of course, much of this program could be created using the Lapidary graphical editor without writing any code, but we will assume here that Lapidary is not being used, and the programmer wants to write everything by hand.

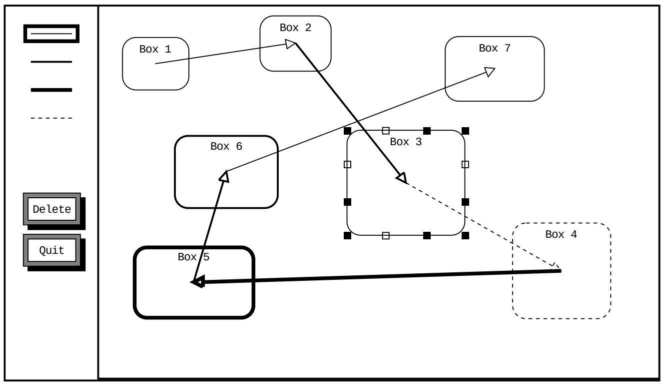


Figure 7:
A simple editor. `Box 3` has been selected by the user, and the current line-style (shown on the left) is a thin line.

To implement this in Garnet, the programmer would first create prototypes for the two kinds of objects that can be created: an arrow, and an aggregate containing a rounded-rectangle and a text object. The aggregate will contain constraints that keep the text centered at the top of the rounded rectangle. Then, a main window would be created containing the buttons for delete and quit, and four line objects to serve as a palette. A rectangle would be added to show which line style is selected, and a menu-interactor would be attached to the four lines with the rectangle as the feedback.

To allow new objects to be created, a `New-Point-Interactor` would be added to the right part of the window which starts on the right button. A parameter to this interactor is the prototype from which instances will be created. Here, this slot will contain an aggregate containing the prototype box and arrows. Formulas in the prototypes will cause the arrows to have the appropriate end points and the string to have the appropriate value.

To make the objects selectable, it is only necessary to include the pre-defined `selection-handle-widget`, which displays the squares around the objects and allows objects to be resized and moved. Internally, this widget contains many formulas that cause the squares to be attached to the objects at the appropriate places (it works for both boxes and lines). The value of the selection-handle widget is the selected object, which will be accessed by the call-back functions for changing the line-style and delete.

To compare the implementations, we asked a number of people to implement the same editor in different object systems and toolkits. Most of these people were the designers of the toolkits.

One implementation was in GINA++, a research toolkit in C++ from the German National Research Center for Computer Science.[4] The implementation defines classes for the line-style palette items, for the commands for creating and deleting objects, for the graphical objects, and for the editor and its panes. Methods on the graphical objects include setting and accessing the ''to'' and ''from'' objects (for the arrows), drawing, and drawing with dashed outline to serve as a feedback object. Methods for the editor include creating the windows, and accessing and setting the current line-style and the selected object. GINA++ provides a retained-object model, so the programmer does not need to write erase or redisplay methods. Support for selection handles around a rectangular object is built in, but the programmer overrode the selection draw-method for lines to only show handles at the end points. To handle creating new objects, when GINA++ sends the `button_press` message to the background window, the `CreateBox` object is created. This special command object defines methods to handle the incremental feedback when dragging out a new box, and then creating a new rounded-rectangle and a new arrow when the mouse button is released.

CLIM [11] is a popular commercial Lisp toolkit that uses CLOS, the standard Common Lisp object system. Like Garnet, CLIM supplies a retained object model with incremental redisplay (which they call ''streams'') and high-level input handling (called ''translators''). Also, CLIM provides a declarative mechanism for defining the window layout, but not for object definitions, so the programmer wrote draw-methods for the objects and selection handles. The programmer also had to write an event handler for the creation of objects, since there is not an appropriate ''translator.''

In both GINA++ and CLIM, methods are needed for drawing objects, since they cannot be specified declaratively. How the rectangle the text is displayed is hard-wired into the draw method of the box class, and thus it might be harder to modify than in Garnet, especially by interactive programs. Because they do not have constraints, the code must explicitly redraw the lines and the text label when the box is moved, whereas in Garnet this is handled automatically.

As a small measure of whether the Garnet technique is more effective, Figure 8 shows the coding time and size information for seven implementations of the editor in Figure 7. All but the MacApp one was implemented by one of the designers of the toolkit, so you can expect that they knew the systems well. The MacApp implementor was also an expert with his system. Zdrava is an experimental, unfinished system, so the times for it are simply estimates from the designer. Of course, these numbers do not constitute a scientific study, and the other programmers did not know that they were participating in a time test. Furthermore, the example was chosen by the Garnet designer. Still, the data does suggest that graphical programs can be smaller and written faster in Garnet.

System	**Language**	**Time**	**Lines of Code**
Garnet	Common Lisp	2.5 hrs	183 lines
CLIM+Zdrava	Common Lisp	2.5 hrs (est.)	190 (est.)
CLIM	Common Lisp	4.5 hrs	331 lines
MacApp	Object Pascal	9 hrs	1026 lines
GINA++	C++	16 hours	550 lines
LispView	Common Lisp	2 days	500 lines
CLM, GINA	Common Lisp	2 to 3 days	273 lines

Figure 8:
Times and code size to create the editor of Figure 7 using various systems. CLIM and GINA are discussed in the article. MacApp is a commercial product of Apple and LispView is a commercial product of Sun.

8. Modularity

Some people claim that using methods is a better interface to objects because it supports better information hiding. The motivation is that the internal implementation of the object can be more easily changed if the interface is through methods. Therefore some object systems, such as SELF [5], do not allow direct access to any object variables, but only provide access

[4] For more information on GINA++ or CLM/GINA for Lisp, contact Mike Spenke, P.O. Box 1316, D-W-5205 St. Augustin 1, Germany, +49 2241 14-2642, spenke@gmd.de.

through methods. Garnet takes an opposite approach, and the main interface is through the data of objects. However, this can be just as modular.

8.1 Data vs. Methods

In Garnet, an object advertises its input and output slots, and most objects of the same type use the same slots (for example, all graphical objects have `:left`, `:top`, `:width`, `:height`, `:filling-style`, etc.). This corresponds to advertising the exported methods in other object systems. In Garnet, through the use of constraint formulas, objects can transform the parameter values in whatever way is desired. For example, the `Menu-Interactor` sets the `:selected` slot of objects. It is up to the internal constraints in the selected object what this does, if anything. The color, position, or font of the object might have a formula depending on the `:selected` slot, and the Interactor does not care. This interface is just as modular as if the Interactor called a generic `Become-Selected` method.

Although Garnet does not currently provide mechanisms to declare which slots of an object can be used from outside and which are internal, this could easily be added. This would provide the same protection as class-instance models which have public and private methods.

8.2 Constraints vs. Methods

Constraints also contribute to modularity in another way, by fixing a flaw in the conventional, imperative object-oriented model. In the conventional model, to achieve certain types of behavior, the programmer must either explicitly arrange the methods so they execute in the proper order, thus violating the modularity of objects, or else allow the methods to execute in an arbitrary order, thus evaluating methods more times than necessary, and possibly destroying the correctness of the program if the methods commit side-effects. For example, suppose that a programmer wants to keep a box called A centered above two other boxes called B and C (Figure 9). In a conventional system, the programmer might add a message to the move methods in B and C that calls a centering method in A. Later the programmer decides that C should always be 20 pixels to the right of B. The programmer thus expands the move method in B to send a message to the move method in C. Without proper sequencing, the centering method in A may be called twice, once by the move method in A, and once by the move method in B. However, the centering method in A should only be called once, after the methods in both B and C have terminated.

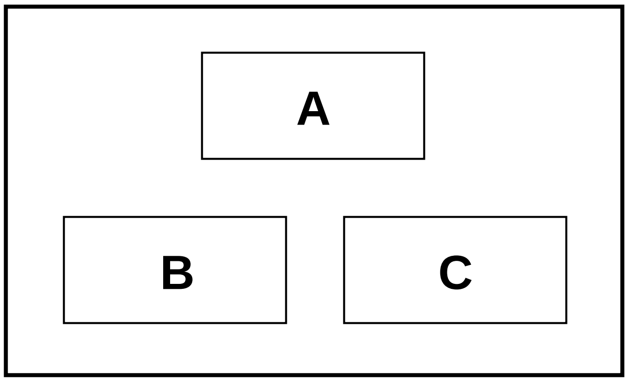


Figure 9:
A box centered over two other boxes. If either box B or C moves, box A should move so that it stays centered over the boxes.

In this case, the programmer is faced with two equally unpalatable choices. The programmer can choose not to provide explicit sequencing, in which case the centering method in A may execute twice. This is both wasteful and potentially dangerous if the centering method commits side-effects (in this case it probably would not, but obviously there are situations where this could pose a problem). Alternatively, the programmer could rely on the fact that the move method in C calls the centering method in A, and thus not call the centering method itself. However, the implementation of the move method in B now depend on the implementation of the move method in C, which violates the notion of modularity.

Notice that in either case the modularity principle is additionally violated because B and C have to know that A depends on them (and later B has to know that C depends on it). If the centering relationship between A, B, and C is later destroyed, not only must the centering method in A be deleted, but the move methods in B and C must be changed as well. (A similar situation arose in the example of Figure 7, where the conventional systems put code in the methods of the boxes to maintain the lines.)

In a constraint-driven language, neither of these problems arises since the constraint solver handles both communication between objects and the ordering of constraints. In the above example, the programmer would initially write a constraint that centered A above B and C. Later the programmer would add an additional constraint placing C 20 pixels to the right of B. The constraint solver would automatically ensure that the constraints were evaluated in the proper order. Thus the programmer would not have to worry about sequencing. In addition, the move methods for B and C would not have to know about the relationships among the three objects (the constraint solver would be responsible for propagating the change information), so they would simply modify the local state of their object. If one or both of the constraints were later deleted, the move methods would not have to be modified. Thus constraint-driven programming better preserves the modularity of objects.

8.3 Interactors vs. Methods

The Garnet input model also provide better modularity than found in other systems. The graphics are entirely independent of the behaviors, and they can be developed and modified separately. In other systems, models, views and controllers have always been tightly coupled, so they all had to be carefully modified together.

8.4 Re-use

Another key feature of Garnet is that it provides better software re-use than most other toolkits. The programmer does not have to re-program new event handlers, since the built-in Interactors are sufficient. The programmer does not need to deal with window refresh or maintaining relationships among objects, since the object system and constraint solver handle this. In addition, since we can be sure that there is an object in memory for every object on the screen, it is possible to provide higher-level widgets, such as the selection-handles. The handles contain constraints that reference the selected object. Toolkits without retained objects cannot supply selection handle widgets because they would need to access the application's internal data structure to know where objects are and how to move and grow the objects.

Another feature of Garnet is that, if the programmer wants to make a slight modification of an existing object, it is only necessary to specify the specific changes to the graphics, rather than having to write completely new draw methods.

9. Conclusion

The style of programming in the Garnet object system is quite different from other object systems: the programmer collects together graphical objects, writes constraints to define the relationships among them, and then attaches instances of pre-defined Interactor objects to cause the objects to respond to the user. Usually, much of the ‘‘programming’’ can be done with graphical, interactive tools, rather than by writing code. Even when not using interactive tools, programmers rarely write methods when creating Garnet code. Our experience suggests that this style of programming is much more effective for graphical user interfaces. It would be interesting to see which other types of programming it works well for. For example, object-oriented data bases seem like a good candidate, since they clearly use a ‘‘retained-object model,’’ and a primary use of methods there is to update objects and to maintain consistency among various objects. Many other application areas might also benefit from this style of programming.

Acknowledgements

For help with this paper, we would like to thank Chris Laffra, Francesmary Modugno, Andrew Mickish, Scott McKay, Jade Goldstein, James Landay, and Bernita Myers. Thanks also to Hans Muller, Scott McKay, Christian Beilken, Markus Sohlenkamp, and John Pane for implementing the example application in different systems and for helping me understand their code.

This research was sponsored by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

References

1. Paul Barth. "An Object-Oriented Approach to Graphical Interfaces". *ACM Transactions on Graphics 5*, 2 (April 1986), 142-172.

2. Alan Borning. "The Programming Language Aspects of Thinglab; a Constraint-Oriented Simulation Laboratory". *ACM Transactions on Programming Languages and Systems 3*, 4 (Oct. 1981), 353-387.

3. Alan Borning and Robert Duisberg. "Constraint-Based Tools for Building User Interfaces". *ACM Transactions on Graphics 5*, 4 (Oct. 1986), 345-374.

4. Paul R. Calder and Mark A. Linton. Glyphs: Flyweight Objects for User Interfaces. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'90, Snowbird, Utah, Oct., 1990, pp. 92-101.

5. Craig Chambers, David Ungar, and Elgin Lee. "An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes". *Sigplan Notices 24*, 10 (Oct. 1989), 49-70. ACM Conference on Object-Oriented Programming; Systems Languages and Applications; OOPSLA'89.

6. James D. Foley and Victor L. Wallace. "The Art of Natural Graphic Man-Machine Conversation". *Proceedings of the IEEE 62*, 4 (April 1974), 462-471.

7. Scott E. Hudson and Shamim P. Mohamed. "Interactive Specification of Flexible User Interface Displays". *ACM Transactions on Information Systems 8*, 3 (July 1990), 269-288.

8. 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 1*, 3 (Aug. 1988), 26-49.

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

10. Mark A. Linton, John M. Vlissides and Paul R. Calder. "Composing user interfaces with InterViews". *IEEE Computer 22*, 2 (Feb. 1989), 8-22.

11. Scott McKay. "CLIM: The Common Lisp Interface Manager". *Comm. ACM 34*, 9 (Sept. 1991), 58-59.

12. Jon Meads. "The Standards Factor". *SIGCHI Bulletin 19*, 1 (July 1987), 34-35.

13. Kenneth J. Meltsner. "A Metallurgical Expert System for Interpreting FEA". *Journal of Metals 43*, 10 (Oct. 1991), 15-17.

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

15. 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, Nov., 1989, pp. 95-104.

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

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

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

19. Brad A. Myers. Graphical Techniques in a Spreadsheet for Specifying User Interfaces. Human Factors in Computing Systems, Proceedings SIGCHI'91, New Orleans, LA, April, 1991, pp. 243-249.

20. Brad A. Myers, Dario Giuse, Roger B. Dannenberg, Brad Vander Zanden, David Kosbie, Philippe Marchal, Ed Pervin, Andrew Mickish, James A. Landay, Richard McDaniel, and Vivek Gupta. The Garnet Reference Manuals: Revised for Version 2.0. Tech. Rept. CMU-CS-90-117-R2, Carnegie Mellon University Computer Science Department, May, 1992.

21. Brad A. Myers and Brad Vander Zanden. "Environment for Rapid Creation of Interactive Design Tools". *The Visual Computer; International Journal of Computer Graphics 8*, 2 (Feb. 1992), 94-116.

22. Lynn Andrea Stein, Henry Lieberman, and David Ungar. A Shared View of Sharing: The Treaty of Orlando. In Won Kim and Frederick H. Lochovsky, Ed., *Object-Oriented Concepts, Databases, and Applications*, ACM Press, Addison-Wesley, New York, NY, 1989, pp. 31-48.

23. Pedro A. Szekely and Brad A. Myers. "A User Interface Toolkit Based on Graphical Objects and Constraints". *Sigplan Notices 23*, 11 (Nov. 1988), 36-45. ACM Conference on Object-Oriented Programming; Systems Languages and Applications; OOPSLA'88.

24. 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.

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