Garnet Tutorial

Andrew Mickish

December 1994

Abstract

This tutorial has been designed to introduce the reader to the basic concepts of Garnet. The reader should have already taken the Garnet Tour before starting the tutorial.

This research was sponsored by NCCOSC under Contract No. N66001-94-C-6037, Arpa Order No. B326. 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 NCCOSC or the U.S. Government.

1. Setting Up

1.1. Take the Tour

Before beginning this tutorial, you should have already completed the Garnet Tour, available in a separate document. The Tour was a series of exercises intended to acquaint you with a few of the features of Garnet, while giving you a feel for the interactive programming aspects of Garnet. This Tutorial investigates all of those features in greater depth, while explaining the fundamental principles behind objects, inheritance, constraints, interactors, and the actual writing of code.

In the Garnet Tour, you were given some background information about how to load Garnet, how to access the different Garnet packages, garbage collection, the main-event-loop for interactors, etc. It may be helpful to review this information from the first few sections of the Tour before starting the Tutorial.

1.2. Load Garnet

Using the instructions from the Tour, load Garnet into your lisp process. Also, type in the following line so that references to the KR package can be eliminated (we will explicitly reference the other Garnet packages):

```
(use-package :KR)
```

2. The Prototype-Instance System

The basic idea behind programming in Garnet is creating objects and displaying them in windows on the screen. An object is any of the fundamental data types in Garnet. Lines, circles, aggregates and windows are all objects. These are all *prototype* objects -- you make copies of these objects and customize the copies to have your desired size and position, as well as other graphic qualities such as filling styles and line styles. When you make a customized copy of an object, we say you create an *instance* of the object. Thus, Garnet is a prototype-instance system.

2.1. Inheritance

When instances are created, an inheritance link is established between the prototype and the instance. *Inheritance* is the property that allows instances to get values from their prototypes without specifying those values in the instances themselves. For example, if we set the filling style of a rectangle to be gray, and then we create an instance of that rectangle, then the instance will also have a gray filling style. Naturally, this leads to an inheritance hierarchy among the objects in the Garnet system. In fact, there is one root object in Garnet -- the `view-object` -- that all other objects are instances of (except for interactors, which have their own root). Figure 2-1 shows some of the objects in Garnet and how they fit into the inheritance hierarchy. (The average user will never be concerned with the actual `view-object` or `graphical-object` prototypes.)

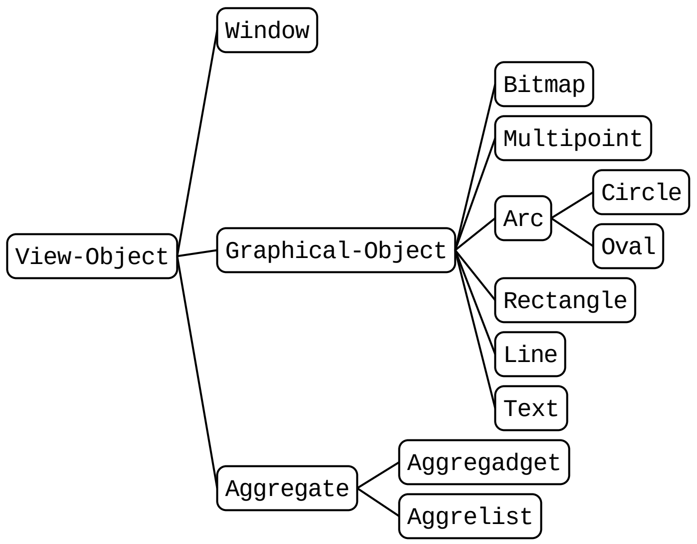

Figure 2-1: The inheritance hierarchy among some of the Garnet prototype objects. All of the standard shapes in garnet are instances of the `graphical-object` prototype. As an example of inheritance, the `circle` and `oval` objects are both special types of arcs, and they inherit most of their properties from the `arc` prototype object. The Gadgets (the Garnet widgets) are not pictured in this hierarchy, but most of them are instances of the `aggregadget` object.

To see an example of inheritance, let's create an instance of a window and look at some of its inherited values. After you have loaded Garnet, type in the following code.

```
(create-instance 'MY-WIN inter:interactor-window
   (:left 800) (:top 100))
(opal:update MY-WIN)   ; To make the window appear
```

The window should appear in the upper-right corner of your screen. In the definition of the MY-WIN schema, we gave a value of 800 to the `:left` slot and a value of 100 to the `:top` slot. Let's check these slots in MY-WIN to see if they are correct. Type in the following lines.

```
(gv MY-WIN :left)     ; Should be 800
(gv MY-WIN :top)      ; Should be 100
```

The function `gv` gets the values of slots from an object. If you got the right values for the `:left` and `:top` slots of MY-WIN, then you see that the values you supplied during the `create-instance` call are still being used by MY-WIN. These are values that are held in the instance itself. On the other hand, try typing in the following lines.

```
(gv MY-WIN :width)
(gv MY-WIN :height)
```

We did not supply values to the `:width` and `:height` slots of MY-WIN when it was created. Therefore, these values are *inherited* from the prototype. That is, they were defined in the `interactor-window` object when it was created, and now MY-WIN inherits those values as its own. We could, however, override these inherited values. Let's change the width and height of MY-WIN using `s-value`, the function that sets the values of slots.

```
(s-value MY-WIN :width 100)
(s-value MY-WIN :height 400)
(opal:update MY-WIN)   ; To cause the changes to appear
```

The dimensions of the window should change, reflecting the new values we have supplied to its `:width` and `:height` slots. If we were to now use `gv` to look at the width and height of MY-WIN, we would get back the new values, since the old ones are no longer inherited.

The inheritance hierarchy which was partially pictured in Figure 2-1 is traced from the leaves toward the root (from right to left) during a search for a value. Whenever we use `gv` to get the value of a slot, the object first checks to see if it has a local value for that slot. If there is no value for the slot in the object, then the object looks to its prototype to see if it has a value for the slot. This search continues until either a value for the slot is found or the root object is reached. When no inherited or local value for the slot is found, the value NIL is returned (which, by the way, looks just the same as a user-defined local value of NIL for a slot).

Since we are now finished with the example of MY-WIN, let's destroy it so it does not interfere with future examples in this tutorial. Type in the following line.

```
(opal:destroy MY-WIN)
```

2.2. Prototypes

When programming in Garnet, inheritance among objects can eliminate a lot of duplicated code. If we want to create several objects that look similar, we could create each of them from scratch and copy all the values that we need into each object. However, inheritance allows us to define these objects more efficiently, by creating several similar objects as instances of a single prototype.

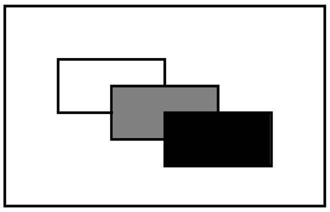

Figure 2-2: Three instances created from one prototype rectangle.

To start, look at the picture in Figure 2-2. We are going to define three rectangles with three different filling styles and put them in a window. First, let's create a window with a top-level aggregate. (For now, just think of an aggregate as an object which contains several other objects.) As we add our objects to this aggregate, they will be displayed in the window.

```
(create-instance 'WIN inter:interactor-window
   (:left 750)(:top 80)(:width 200)(:height 400))
(create-instance 'TOP-AGG opal:aggregate)
(s-value WIN :aggregate TOP-AGG)
(opal:update WIN)
```

Now let's consider the design for the rectangles. The first thing to notice is that all of the rectangles have the same width and height. Therefore, we will create a prototype rectangle which has a width of 40 and a height of 20, and then we will create three instances of that rectangle. To create the prototype rectangle, type the following.

```
(create-instance 'PROTO-RECT opal:rectangle
   (:width 40) (:height 20))
```

This rectangle will not appear anywhere, because it will not be added to the window. But now we need to create the three actual rectangles that will be displayed. Since the prototype has the correct values for the width and height, we only need to specify the left, top, and filling styles of our instances.

```
(create-instance 'R1 PROTO-RECT
   (:left 20) (:top 20)
   (:filling-style opal:white-fill))

(create-instance 'R2 PROTO-RECT
   (:left 40) (:top 30)
   (:filling-style opal:gray-fill))

(create-instance 'R3 PROTO-RECT
   (:left 60) (:top 40)
   (:filling-style opal:black-fill))

(opal:add-components TOP-AGG R1 R2 R3)  ; Give the aggregate three components
(opal:update WIN)
```

After you update the window, you can see that the instances R1, R2, and R3 have inherited their `:width` and `:height` from PROTO-RECT. You may wish to use `gv` to verify this. With these three rectangles still in the window, we are ready to look at another important use of inheritance. Try changing the width and height of the prototype as follows.

```
(s-value PROTO-RECT :width 30)
(s-value PROTO-RECT :height 40)
(opal:update WIN)
```

The result should look like the rectangles in Figure 2-3. Just by changing the values in the prototype rectangle, we were able to change the appearance of all its instances. This is because the three instances inherit their width and height from the prototype, even when the prototype changes.

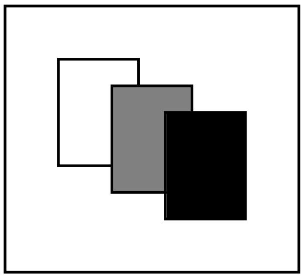

Figure 2-3: The instances change whenever the prototype object changes.

For our last look at inheritance in this section, let's try to override the inherited slots in one of the instances. Suppose we now want the rectangles to look like Figure 2-4. In this case, we only want to change the dimensions of one of the instances. The following lines should change the appearance of the black rectangle accordingly.

```
(s-value R3 :width 100)
(opal:update WIN)
```

The rectangle R3 now has its own value for its `:width` slot, and no longer inherits it from PROTO-RECT. If you change the `:width` of the prototype again, the width of R3 will not be affected. However, the width of R1 and R2 will change with the prototype, because they still inherit the values for their `:width` slots. This shows how inheritance can be used flexibly to make specific exceptions to the prototype object.

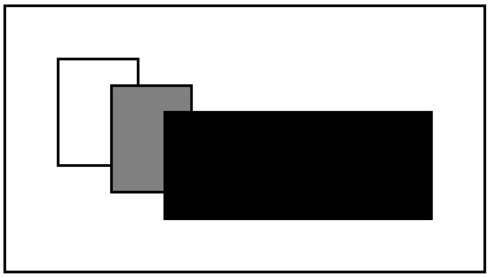

Figure 2-4: The width of R3 is overridden, so it is no longer inherited from the prototype.

2.3. Default Values

Because of inheritance, all instances of Garnet prototype objects have reasonable default values when they are created. As we saw in section 2.1, the `interactor-window` object has its own `:width`. So, if an instance of it is created without an explicitly defined width, the width of the instance will be inherited from the prototype, and it can be considered a default value.

2.4. The Inspector

An important tool for examining properties of objects is the Inspector. This tool is loaded with Garnet by default, and resides in the package `garnet-debug`. The Inspector is described in detail in the Debugging Manual that starts on page 461 of this reference manual.

To run the inspector on our example of three rectangles, position the mouse over R3 (the black rectangle) in the window, and hit the HELP key. If your keyboard does not have a HELP key, or hitting it does not seem to do anything, you can start the Inspector manually by typing `(gd:Inspector R3)` into the lisp listener. The Inspector window that appears will look like figure 2-5.

The local slots and values for R3 are shown in the Inspector window. Inherited slots are not shown, like `:height` or `:line-style` (assuming that you did not set these slots yourself, installing local values in R3). If you have a color screen, some slots are red, indicating that these slots are public "parameters" of the object (we discuss parameters more in section 2.5).

It is very easy to change properties of an object with the Inspector. For example, to change the `:width` of R3 using the inspector, click the mouse on the value of the `:width` slot (which is 100 in figure 2-5). Use standard Emacs commands to change the value of the slot to something significantly different, like 20. When you hit RETURN, the change will appear instantly in R3.

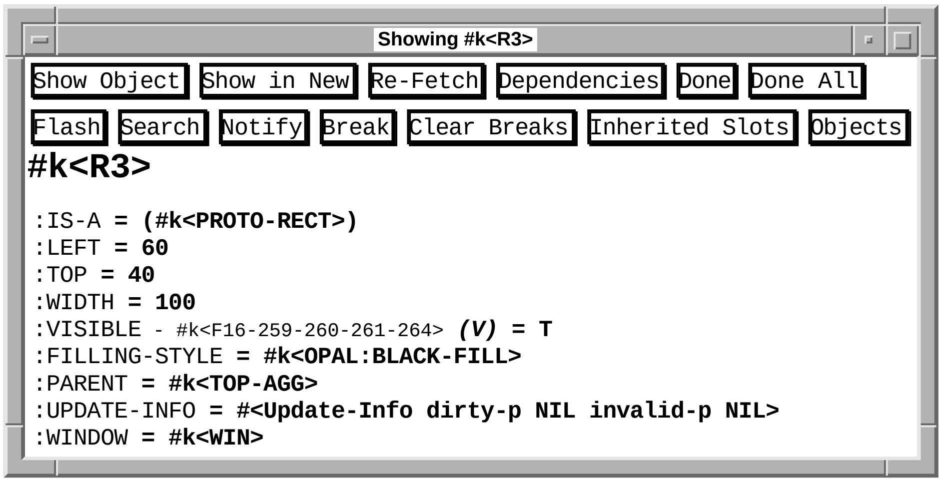

Figure 2-5: The Inspector displaying the slots and values of rectangle R3.

To add a new local value to R3 -- that is, to override an inherited value with a new local value -- you have to add an extra line to the Inspector window. In our example, R3 does not have a local value for `:height`, since its value is inherited from the prototype PROTO-RECT. To override this value, click the cursor at the end of a line, and type `^j` to add a new line to the display. Now you can type "`:height = 100`" and hit RETURN to install the new slot/value pair. The change should be reflected instantly in R3.

You can bring up other Inspector windows by positioning the mouse over another object and hitting HELP again, or you can select text that is already displayed in the Inspector and using the "`Show Object`" or "`Show in New`" buttons. For example, to examine the `opal:black-fill` object that is the value of R3's `:filling-style` slot, either click-and-drag or double-click on the `#k<OPAL:BLACK-FILL>` value and press the "`Show in New`" button. The object will be displayed in a new window.

When you are finished with the Inspector, you can click on the "`Done`" or "`Done All`" buttons to make the Inspector windows disappear.

Significantly more detail about the Inspector is included in the Debugging Manual, including how to explore the Prototype/Instance hierarchy of objects, and how to use the Inspector for debugging more compilcated examples.

2.5. Parameters

Most objects in Garnet have a list of *parameters*, which are stored in the `:parameters` slot. This is a list of all customizable properties of the object. For example, `gv opal:rectangle :parameter` yields:

```
(:LEFT :TOP :WIDTH :HEIGHT :LINE-STYLE :FILLING-STYLE :DRAW-FUNCTION :VISIBLE)
```

These can be considered the "public" slots of `opal:rectangle`, which can be given customized values when instances are created. If values are not supplied for these slots when instances are created, the default values will be inherited from the prototype object.

There are other slots that change when instances of `opal:rectangle` are added to a window, such as the `:window` and `:parent` slots, but these slots are not intended to be set manually. Since they are "read-only" slots, they are not included in the `:parameters` list.

Several tools in Garnet rely heavily on the `:parameters` slot. As discussed in section 2.4, the Inspector displays the parameter slots in red, so that they are easily identified. The `gg:prop-sheet` gadget which is used in Gilt and Lapidary looks at the `:parameters` slot to determine which slots should be displayed for the user to customize. These objects are discussed thoroughly in later sections of this reference manual.

The typical Garnet user will not have to worry much about the `:parameters` slot. All of the slots that are in the list are documented in this manual, so it is really just another way to access the same information about properties of objects. For details on defining `:parameters` slots for your own objects, see the KR Manual. Unless you are defining your own list for a special object, the `:parameters` slot should be considered read-only.

2.6. Destroying Objects

Before moving on to the next section, destroy the window so that it does not interfere with future examples in this tutorial. Type the following line.

```
(opal:destroy WIN)
```

Destroying the window will also destroy all of the objects that were added to its aggregate. We can no longer manipulate R1, R2, and R3, since they were destroyed by the previous call. However, the PROTO-RECT was never added to the top-level aggregate, and it was not destroyed. You could destroy this object now with a `destroy` call, but we will be using this object again in Section 2.6. So, leave the object residing in memory for now.

When an object is destroyed, its variable name becomes unbound and the memory space that was allocated to the object is freed. You can `destroy` any object, including windows. If you destroy a window, all objects inside of it are automatically destroyed. Similarly, if you destroy an aggregate, all objects in it are destroyed. When you destroy a graphical object (like a line or a circle), it is automatically removed from any aggregate it might be in and erased from the screen.

If a prototype object is destroyed (i.e., an object that has had instances created from it), then all of the instances of that object will be recursively destroyed.

Occasionally in the course of developing a program, you may (either accidentally or intentionally) define a new object which happens to have the same name as an old object. When the new object is created, its variable name is set to the new object, and the old object by the same name is destroyed. Also, all of the instances of the old object are recursively destroyed.

For example, in Section 2.2 above, we created the object PROTO-RECT, which still exists in memory. If we now enter the following new schema definition for an object by the same name, then the old PROTO-RECT will be destroyed.

```
(create-instance 'PROTO-RECT opal:rectangle)
```

When the new schema is entered, a warning is given that the old object is being destroyed. You can safely ignore this message, assuming that you intended to override the definition of the old schema.

2.7. Unnamed Objects

Sometimes you will want to create objects that do not have a particular name. For example, you may want to write a function that returns a rectangle, but it will be called repeatedly and should not destroy previous instances with new ones. In this case, you should return an unnamed rectangle from the function which can be used just like the named objects we have created earlier in this tutorial.

As an example, the following code creates an unnamed object and internally generates a unique variable name for it. Instead of supplying a quoted name to `create-instance`, we give it the value NIL.

```
(create-instance NIL opal:rectangle
   (:left 10) (:top 10) (:width 75) (:height 50))
```

When you enter this schema definition, the `create-instance` call will return the generated internal name of the rectangle -- something like `RECTANGLE-123`. This name has a unique number as a suffix that prevents it from being confused with other rectangles in Garnet. You can now use the generated name to refer to the object.

```
(gv RECTANGLE-123 :top)   ; Replace this name with the name of your rectangle.
```

Usually it is convenient to assign an unnamed object to a local variable. The following line creates a circle and assigns it to the new variable MY-CIRCLE.

```
(setf MY-CIRCLE (create-instance NIL opal:circle))
```

Now MY-CIRCLE will have the generated circle as its value. If the same line were entered again, the old circle would not be destroyed, but the variable MY-CIRCLE would still point to a new one. This can be useful inside a function that uses a `let` clause -- every time the `let` is executed, new objects are assigned to the local variables, but the old objects still remain in memory and are not destroyed. Section 6.2 contains an example of how unnamed objects might be used in a function.

3. An Overview of the Objects

3.1. Lines, Rectangles, and Circles

The Opal package provides different graphical shapes including circles, rectangles, roundtangles, and lines. There are also several different kinds of text, and some special objects like bitmaps and arrowheads. Each graphical object has special slots that determine its appearance, which are documented in the Opal manual. (For example, the line uses the slots `:x1`, `:y1`, `:x2`, and `:y2`.) See the section "Specific Graphical Objects" in the Opal manual for details of how each object works. Examples of creating instances of graphical objects appear throughout this tutorial.

3.2. Aggregates

In order to put a large number of objects into a window, we might create all of the objects and then add them, one at a time, to the window. However, this is usually not how we organize the objects conceptually. For example, if we were to create a sophisticated interface with a scroll bar, several buttons, and labels for the buttons, we would not want to add each rectangle in the scroll bar and the buttons individually. Instead, we would think of creating the scroll bar from its composite rectangles, then creating the buttons along with their labels, and then adding the scroll bar assembly and the button assembly to the window.

Grouping objects together like this is the function of the `aggregate` object. Any graphical object can be a component of an aggregate - lines, circles, rectangles, and even other aggregates. Usually all of the components of an aggregate are related in some way, like they are all parts of the same button.

Two other objects, the `aggregadget` and the `aggrelist`, are also used to group objects, and usually appear more often in Garnet programs. `Aggregadgets` and `aggrelists` are instances of `aggregate`, and they have special features that make them very useful in creating objects. These objects will be discussed further in Section 3.3.

The top-level object in a window is always an aggregate. This aggregate contains all of the objects in that window. Therefore, for an object to appear in a window it either has to be a component of the top-level aggregate, or it has to be a component of another aggregate which, at the top of its aggregate hierarchy, is a component of the top-level aggregate.

When aggregates have other aggregates as components, an aggregate hierarchy is formed. This hierarchy describes the way that objects are grouped together. Figure 3-1 shows how the objects that comprise a vertical scroll bar might be conceptually organized.

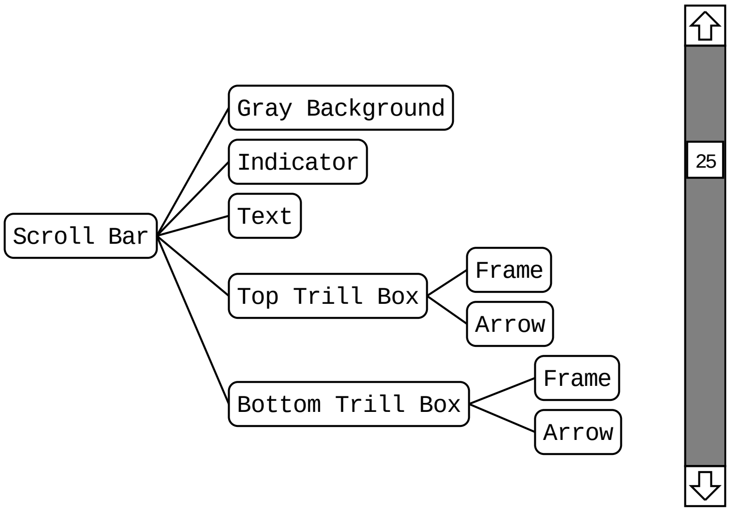

Figure 3-1: One possible hierarchy for the objects that make up a scroll bar.

In the scroll bar hierarchy, all of the leaves correspond to shapes that appear in the scroll bar. The leaves are always Opal graphic primitives, like rectangles and text. The nodes `top-trill-box` and `bottom-trill-box` are both aggregates, each with two components. And, of course, the top-level `scroll-bar` node is an aggregate.

This aggregate hierarchy should not be confused with the inheritance hierarchy that was discussed earlier. Components of an aggregate do not inherit values from their parents. Instead, relationships among aggregates and components must be explicitly defined using constraints, a concept which will be discussed shortly in this tutorial.

When an object is added to an aggregate, its `:parent` slot is set to point to that aggregate. Therefore, in Figure 3-1, the `:parent` of the `bottom-trill-box` is the `scroll-bar` aggregate. This `:parent` slot is called a *pointer* slot because its value is another Garnet object. Pointer slots are discussed further in section 3.3.

The functions `add-component`, `remove-component`, and `move-component` are used to manipulate the components of an aggregate. Descriptions of these and other functions for components may be found in the "Aggregate Objects" section of the Opal manual.

3.3. Aggregadgets, Aggrelists, and Aggregraphs

Aggregadgets and aggrelists are types of aggregates. With these objects, an aggregate and its components can basically be defined simultaneously. In aggregadgets, all the components are defined with a list in the `:parts` slot. In an aggrelist, a single object is defined to be an "item-prototype", and the aggrelist automatically generates several instances of that object to make its components. The aggregraph is a type of aggregadget, where all the components are nodes and arcs that make up a graph. Figures 2-1 and 3-1 were created using Garnet aggregraphs. For several examples and a complete discussion of how to use aggregraphs, see the Aggregadgets, Aggrelists, and Aggregraphs Reference Manual.

3.3.1. Aggregadgets

When you create an aggregadget, you may list all of the objects that you want as components of the aggregadget in the `:parts` slot. The list is specified using the standard Lisp backquote macro, and there are usually many function calls and objects inside the list that must be evaluated with a comma. As an example of an aggregadget, we will analyze the following schema definition, but it is not necessary to type it in. This code contains a few references that have not been discussed in this tutorial yet, but it serves the purpose of giving us a plain aggregadget to study.

```
(create-instance 'AGG opal:aggregadget
   (:left 10) (:top 20)
   (:parts
    `((:my-circle ,opal:circle
                  (:left 60) (:top 70)
                  (:width 100) (:height 100)
                  (:line-style ,opal:dashed-line))
      (:my-rect ,opal:rectangle
                (:left ,(o-formula (gvl :parent :left)))
                (:top ,(o-formula (gvl :parent :top)))
                (:width 80) (:height 40)
                (:filling-style ,opal:black-fill)
                (:line-style NIL)))))
```

The `:parts` slot in the AGG object contains a list of lists, with each internal list being a definition of a component. The components of AGG will be a circle and a rectangle, to which we have given the arbitrary names `:my-circle` and `:my-rect`. These names, which are preceded by a colon, will be the names of new slots in the aggregadget. That is, two *pointer* slots will be created in AGG, named `:my-circle` and `:my-rect`, which will have the circle and rectangle objects as their values. We say these are pointer slots because they point to other objects.

Other pointer slots which are automatically created are the `:parent` slots of both the circle and the rectangle. Since these objects are being added as components to the aggregadget, their `:parent` slots are set as with aggregates. Thus, a two-way path of communication is established between the aggregadget and each of its components -- the `:parent` slot points up, and the `:my-circle` slot points down.

Notice that the `:parts` list is backquoted (with a ` instead of a '). Using this backquote syntax, we can then use commas to evaluate the names of objects inside the list. The references to be evaluated are the two graphical object prototypes (the `opal:circle` and the `opal:rectangle`) and the graphical qualities (`opal:dashed-line` and `opal:black-fill`). Commas are also used to evaluate the `o-formula` calls, which establish constraints among objects (constraints are discussed in Chapter 4). If the commas were not present inside the `:parts` list, then the names of all the Garnet objects would not be dereferenced, and they would be treated as mere atoms, not objects. Similarly, the calls to `o-formula` would appear as simple quoted lists instead of function calls.

An important difference between aggregates and aggregadgets is that when you create an instance of an aggregadget, the instance will automatically have components that match those in the prototype. That is, if we created an instance of AGG, called AGG-INSTANCE, then AGG-INSTANCE would automatically have a circle and rectangle component just like AGG. In contrast, when you create an instance of an aggregate, the components are not automatically generated, and you would have to create and add them to the instance manually.

Other examples of aggregadget definitions can be found in sections 4.5 and 6.1 in this tutorial.

3.3.2. Aggrelists

An aggrelist allows you to create and easily arrange objects into a nicely formatted graphical list. The motivation for aggrelists comes from the arrangement of groups of objects like button panels, tic-marks, and menu choices, where all the components of an aggregate are similar and should appear in a vertical or horizontal list.

In an aggrelist, a single item-prototype is defined, and then this object is automatically copied several times to make the components of the aggrelist. The `:left`, `:top`, and other slots of the components are automatically given values that will neatly lay out the components in a list, so that the programmer does not have to do any calculations for the positions of the objects.

As with aggregadgets, aggrelists use the backquote syntax to define the item prototype. There are many customizable aspects of aggrelists, such as whether to orient the components vertically or horizontally, the distance between each object, etc. Since there are so many customizable slots, please see the Aggregadgets and Aggrelists Reference Manual for a discussion of how to use aggrelists. Section 6.1 in this tutorial includes an example of the definition of an aggrelist.

3.4. Windows

When we want to add an object to a window, what we really mean is that we want to add the object to the window's top-level aggregate (or to an aggregate at a lower level in that window's aggregate hierarchy). Every window has one top-level aggregate, and all objects that appear in the window are components in its aggregate hierarchy.

Any object must be added to a window in order for it to be shown on the screen. Additionally, a window must be updated before any changes made to it (or the objects in it) will appear. Windows are updated when you explicitly issue a call to `opal:update`, and they are also continuously updated when interactors are running and changing objects in the window (during the `main-event-loop`, discussed in Section 5).

3.5. Gadgets

The Garnet gadgets are a set of ready-made widgets that can be treated as regular graphical objects. They have slots that can be customized with user-defined values, and are added to windows just like graphical objects. Generally, they are objects that are commonly found in an interface including scroll bars, menus, buttons and editable text fields. In the Tour, you created instances of the radio button panel and the vertical slider. There are also more sophisticated gadgets like scrolling windows, property sheets (to allow quick editing of the slots of objects), and selection handles (for moving and growing objects).

Most of the gadgets come in two versions -- one called the Garnet Style, and one modeled after the OSF/Motif style. Examples of how to use the gadgets are found in demonstration programs at the end of each of the gadget files, which can be executed by commands like `(garnet-gadgets:menu-go)`. For detailed descriptions of all the available gadgets, see the Gadgets Reference Manual.

4. Constraints

In the course of putting objects in a window, it is often desirable to define relationships among the objects. For example, you may want the tops of several objects to be aligned, or you might want a set of circles to have the same center, or you may want an object to change color if it is selected. Constraints are used in Garnet to define these relationships among objects.

Although all the examples in this section use constraints on the positions of objects, it should be clear that constraints can be defined for filling styles, strings, or any other property of a Garnet object. Many examples of constraints can be found in other sections of this tutorial. Additionally, much of the KR Reference Manual is devoted to the discussion of constraints among objects. The sections "Constraint Maintenance" and "Slot and Value Manipulation Functions" should be of particular interest.

4.1. Formulas

A formula is an explicit definition of how to calculate the value for a slot. If we want to constrain the top of one object to be the same as another, then we define a formula for the `:top` slot of the dependent object. With constraints, the value of one slot always *depends* on the value of one or more other slots, and we say the formula in that slot has *dependencies* on the other slots.

An important point about constraints is that they are constantly maintained by the system. That is, they are evaluated once when they are first created, and then they are continually *re-evaluated* when any of their dependencies change. Thus, if several objects depend on the top of a certain rectangle, then all the objects will change position whenever the rectangle is moved.

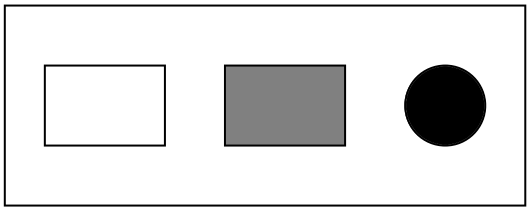

Figure 4-1: Three objects that are all aligned with the same top. The top of the gray rectangle is constrained to the white rectangle, and the top of the black circle is constrained to the top of the gray rectangle.

As our first example of defining constraints among objects, we will make the window in Figure 4-1. Let's begin by creating the white rectangle at an absolute position, and then create the other objects relative to it. Create the window and the first box with the following code.

```
(create-instance 'CONSTRAINTS-WIN inter:interactor-window      ; Create the window
   (:left 750)(:top 80)(:width 260)(:height 100))
(create-instance 'TOP-AGG opal:aggregate)                      ; Create an aggregate
(s-value CONSTRAINTS-WIN :aggregate TOP-AGG)                   ; Assign the aggregate to the window
(opal:update CONSTRAINTS-WIN)                                  ; Make the window appear

(create-instance 'WHITE-RECT opal:rectangle                    ; Create a rectangle
   (:left 20) (:top 30)
   (:width 60) (:height 40)
   (:filling-style opal:white-fill))

(opal:add-components TOP-AGG WHITE-RECT)                       ; Add the rectangle to the window
(opal:update CONSTRAINTS-WIN)                                  ; Make changes in the window appear
```

We are now ready to create the other objects that are aligned with WHITE-RECT. We could simply create another rectangle and a circle that each have their top at 30, but this would lead to extra work if we ever wanted to change the top of all the objects, since each object's `:top` slot would have to be changed individually. If we instead define a relationship that depends on the top of WHITE-RECT, then whenever the top of WHITE-RECT changes, the top of the other objects will automatically change, too. Define the schema for the gray rectangle as follows.

```
(create-instance 'GRAY-RECT opal:rectangle
   (:left 110)
   (:top (o-formula (gv WHITE-RECT :top)))   ; Constrain the top of this rectangle to the top of WHITE-RECT
   (:width 60) (:height 40)
   (:filling-style opal:gray-fill))

(opal:add-components TOP-AGG GRAY-RECT)
(opal:update CONSTRAINTS-WIN)
```

You can see that without specifying an absolute position for the top of the gray rectangle, we have constrained it to always have the same top as the white rectangle. The formula in the `:top` slot of the gray rectangle was defined using the functions `o-formula` and `gv`. The `o-formula` function is used to declare that an expression is a constraint. When `gv` is used inside a formula, it causes a dependency to be established on the referenced slot, so that the formula will be reevaluated when the value in the referenced slot changes.[1]

To see if our constraint is working, try changing the top of the white rectangle with the following instructions and notice how the gray rectangle moves with it. Try setting the top to other values, if you wish.

```
(s-value WHITE-RECT :top 50)
(opal:update CONSTRAINTS-WIN)
```

The important thing to notice is that the value of the `:top` slot of GRAY-RECT changes as the top of the WHITE-RECT changes. This shows that the formula in GRAY-RECT is being re-evaluated whenever its depended values change.

Now we are ready to add the black circle to the window. We have a choice of whether to constrain the top of the circle to the white rectangle or the gray rectangle. Since we are going to be examining these objects closely in the next few paragraphs, let's constrain the circle to the gray rectangle, resulting in an indirect relationship with the white one. Define the black circle with the following code.

```
(create-instance 'BLACK-CIRCLE opal:circle
   (:left 200)
   (:top (o-formula (gv GRAY-RECT :top)))
   (:width 40) (:height 40)
   (:filling-style opal:black-fill))

(opal:add-components TOP-AGG BLACK-CIRCLE)
(opal:update CONSTRAINTS-WIN)
```

At this point, you may want to set the `:top` of the white rectangle again just to see if the black circle follows along with the gray rectangle.

4.2. Cached Values

An interesting question might have occurred to you -- what happens if you set the `:top` of the gray rectangle now? Setting the value of a slot which already has a formula in it does <u>not</u> destroy the existing constraint. However, it does override the current *cached* value of the formula. Try setting the `:top` of the gray rectangle now.

[1]There is another function called `g-value` that is similar to `gv`, except that it never causes dependencies to be established. Older versions of Garnet required that `gv` only be used inside formulas, and `g-value` to be used ouside. The `gv` function has since been enhanced so that it can be used everywhere. It would be unusual to ever need to use `g-value`.

```
(s-value WHITE-RECT :top 30)   ; Return everything to its original position
(opal:update CONSTRAINTS-WIN)

(s-value GRAY-RECT :top 40)
(opal:update CONSTRAINTS-WIN)
```

The position of WHITE-RECT will remain unchanged, since it was defined with an absolute position. However, the new value that we gave for the top of the gray rectangle has repositioned GRAY-RECT and BLACK-CIRCLE. Previously, the formula in the `:top` slot of GRAY-RECT had correctly computed its own top, getting the value from the `:top` slot of WHITE-RECT. Now, however, we replaced that cached value with our absolute value of 40.

To show that the formula is still alive and well in the `:top` slot of GRAY-RECT, try setting the `:top` slot of WHITE-RECT again.

```
(s-value WHITE-RECT :top 10)
(opal:update CONSTRAINTS-WIN)
```

Since the top of GRAY-RECT depends on WHITE-RECT, its formula will be recomputed whenever the top of WHITE-RECT changes. There is now a new cached value for the `:top` of GRAY-RECT, a result of re-evaluating the formula.

4.3. Formulas and s-value

It is important to distinguish the behavior of `s-value` when it is used on a slot with a formula in it, versus using it on a slot with an absolute value in it (like the number 5). Setting the value of a slot that *already* has a formula in it will not destroy the old formula. Instead, only the cached value of the formula is changed, and the formula will be re-evaluated if any of its dependencies change.

On the other hand, `s-value` will replace one absolute value with another absolute value, and the old value will never appear again. That is, if an object was created with some particular absolute value for a slot, and we changed that slot with `s-value`, then the new value will be permanent until the slot is explicitly set again with `s-value`.

The one exception to the above rules is when the new value is a formula itself. Using `s-value` to set a new formula will always obliterate what was previously in the slot, whether it was an absolute value or a formula.

4.4. Using the :obj-over Slot

When designing an interface, you may want a box to be drawn around an object to show that it is selected. In the usual case, you will want to define only one box that will be drawn around different objects, and it would be nice if the box changed size when it was over objects of different size. The traditional Garnet approach to this problem is to use constraints in the dimension slots of the selection box that depend on the dimensions of the object it is over.

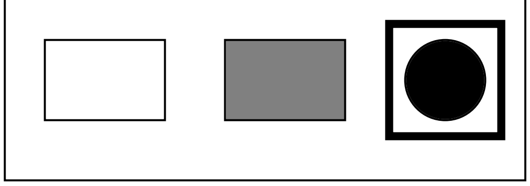

Figure 4-2: A selection box drawn around an object.

In the traditional approach, we use the slot `:obj-over` in the selection box to specify which object the selection box should be drawn around. The `:obj-over` slot is a pointer slot, since it contains an object as its value (pointer slots were discussed in section 3.3). Then, we define formulas for the dimensions of the selection box which depend on the `:obj-over` slot. The formulas in the following schema definition should be clear.

```
(create-instance 'SEL-BOX opal:rectangle
   (:obj-over GRAY-RECT)  ; A pointer slot
   (:left (o-formula (- (gvl :obj-over :left) 10)))
   (:top (o-formula (- (gvl :obj-over :top) 10)))
   (:width (o-formula (+ 20 (gvl :obj-over :width))))
   (:height (o-formula (+ 20 (gvl :obj-over :height))))
   (:line-style opal:line-4))  ; A line with a thickness of 4 pixels

(opal:add-components TOP-AGG SEL-BOX)
(opal:update CONSTRAINTS-WIN)
```

Now if you set the `:obj-over` slot of the selection box to be a different object, the position and dimensions of SEL-BOX will change according to the object.

```
(s-value SEL-BOX :obj-over BLACK-CIRCLE)
(opal:update CONSTRAINTS-WIN)
```

You may have noticed that we performed computations in the formulas above, instead of just using values directly as in the GRAY-RECT and BLACK-CIRCLE objects. In fact, formulas can contain any lisp expression. Also, the formulas above use the function `gvl` instead of `gv`, which was used earlier. We use `gvl` here because we are referencing a slot, `:obj-over`, in the same schema as the formula. Previously, we used `gv` to look at a slot in a different object.

Another point of interest in these formulas is the use of indirect references. In the `:left` formula above, the pointer slot `:obj-over` is referenced, and then its `:left` slot is referenced, in turn. It is important to distinguish between this case and the case where a value is stored in the same object as the formula, as in the following example.

Suppose we want the selection box to become invisible if no object is currently selected. Using `s-value`, we can set the `:visible` slot of SEL-BOX to have a formula which will cause the box to disappear when its current selection is NIL. (We could have also defined this formula in the `create-instance` call.)

```
(s-value SEL-BOX :visible (o-formula (gvl :obj-over)))
```

Clearly, if `:obj-over` is set to NIL, then the value in the `:visible` slot will also become NIL, and the box will become invisible. When `:obj-over` is again set to be some object, then the `:visible` slot will have a non-NIL value, and the box will appear in the appropriate position. Previously, if we had set `:obj-over` to NIL, then updating the window would have caused an error when the formulas tried to access the `:left`, `:top`, `:width`, and `:height` of the non-existent selected object.

We are finished with the objects from this section, but the next section will continue to use the same window. So, to remove the old objects from the window, use the function `remove-components`.

```
(opal:remove-components TOP-AGG WHITE-RECT GRAY-RECT BLACK-CIRCLE SEL-BOX)
(opal:update CONSTRAINTS-WIN)
```

4.5. Constraints in Aggregadgets

As mentioned in section 3.2, the `:parent` slot of a component is automatically set to its parent aggregate when it is attached. Since aggregadgets are instances of aggregates, all of the components defined in an aggregadget have their `:parent` slot set in this way. In this section, we will examine how this slot can be used to communicate between components of an aggregadget.

The aggregadget we will use in this example will make the picture of concentric circles in Figure 4-3. Suppose that we want to be able to change the size and position of the circles easily, and that this should be done by setting as few slots as possible.

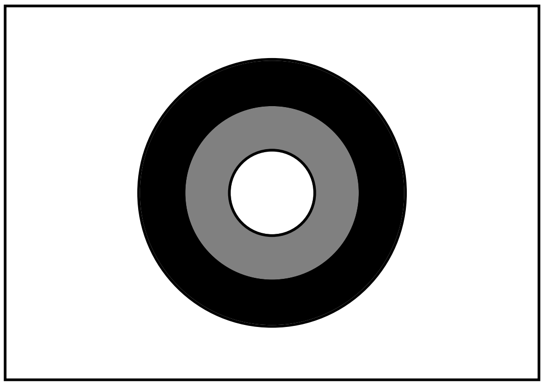

Figure 4-3: An aggregadget with three circles as components.

From the picture, we see that the dimensions of the black circle are the same as the dimensions of the entire group of objects. That is, if a bounding box were drawn around the black circle, all the other objects would be inside the bounding box, too. Therefore, it will be helpful to put slots for the size and position of this circle in the top-level aggregadget, and have all the circles reference these top-level values through formulas.

To start, let's create an aggregadget with only one component -- the black circle -- and then redefine the aggregadget with the other components later. The following code creates this one-component aggregate.

```
(create-instance 'CON-CIRCLES opal:aggregadget
   (:left 20) (:top 20)
   (:width 100)
   (:height (o-formula (gvl :width)))
   (:parts
    `((:black-circle ,opal:circle
                     (:left ,(o-formula (gvl :parent :left)))
                     (:top ,(o-formula (gvl :parent :top)))
                     (:width ,(o-formula (gvl :parent :width)))
                     (:height ,(o-formula (gvl :width)))
                     (:filling-style ,opal:black-fill)))))

(opal:add-component TOP-AGG CON-CIRCLES)
(opal:update CONSTRAINTS-WIN)
```

All those commas are needed because we want the Opal objects and the calls to `o-formula` to be evaluated inside the backquoted list. If the commas were not present, then those forms would become inert atoms and lists instead of objects and function calls.

The black circle in the aggregadget gets its position and dimensions from the top-level slots in CON-CIRCLES. The communication link used here is the `:parent` slot, which points from the component to the aggregadget. The function `gvl` is used in the formulas for the black circle because the `:parent` slot is in the same object as the formulas. Notice that the black circle does not "inherit" any values from its parent. Creating components in an aggregadget sets up an *aggregate* hierarchy, where values travel back-and-forth over constraints, not inheritance links. If you want a component to depend on values in its parent, you have to define constraints.

The other components of CON-CIRCLES will be defined analogously, but with a little more computation in the formulas to get them to line up properly. Before typing in the new definition of CON-CIRCLES, remove the old aggregadget from the window with the following instruction.

```
(opal:remove-components TOP-AGG CON-CIRCLES)
(opal:update CONSTRAINTS-WIN)
```

And now we are ready to redefine CON-CIRCLES again, this time with an extra top-level slot to reduce redundant calculations, and of course with the other two circles.

```
(create-instance 'CON-CIRCLES opal:aggregadget
   (:left 20) (:top 20)
   (:width 100)
   (:height (o-formula (gvl :width)))
   (:radius/3 (o-formula (round (gvl :width) 6)))
   (:parts
    `((:black-circle ,opal:circle
                     (:left ,(o-formula (gvl :parent :left)))
                     (:top ,(o-formula (gvl :parent :top)))
                     (:width ,(o-formula (gvl :parent :width)))
                     (:height ,(o-formula (gvl :width)))
                     (:filling-style ,opal:black-fill))
      (:gray-circle ,opal:circle
                    (:left ,(o-formula (+ (gvl :parent :left)
                                          (gvl :parent :radius/3))))
                    (:top ,(o-formula (+ (gvl :parent :top)
                                         (gvl :parent :radius/3))))
                    (:width ,(o-formula (- (gvl :parent :width)
                                           (* 2 (gvl :parent :radius/3)))))
                    (:height ,(o-formula (gvl :width)))
                    (:filling-style ,opal:gray-fill))
      (:white-circle ,opal:circle
                     (:left ,(o-formula (+ (gvl :parent :gray-circle :left)
                                           (gvl :parent :radius/3))))
                     (:top ,(o-formula (+ (gvl :parent :gray-circle :top)
                                          (gvl :parent :radius/3))))
                     (:width ,(o-formula (- (gvl :parent :gray-circle :width)
                                            (* 2 (gvl :parent :radius/3)))))
                     (:height ,(o-formula (gvl :width)))
                     (:filling-style ,opal:white-fill)))))

(opal:add-components TOP-AGG CON-CIRCLES)
(opal:update CONSTRAINTS-WIN)
```

The gray circle gets its size and position from the top-level slots just like the black circle, only it is one-third the size. The white circle is the most interesting case, where it gets its position and dimensions from the gray circle. Not only does the white circle communicate with the aggregadget through the `:parent` slot, but it also uses the slot `:gray-circle` which was automatically created in the aggregadget (see section 3.3). Thus, the formulas in the white circle trace up the aggregate hierarchy to the parent aggregadget, and then back down into another component.

To examine these pointer slots more closely, try executing the following line.

```
(gv CON-CIRCLES :white-circle)
```

The value returned by this `gv` call is the internally generated name of the white circle. This name was generated with a unique suffix number so that it will not be confused with some other white circle in Garnet (see Section 2.7). You can also look at slots of the components directly, by adding slot names to the end of the `gv` call, like

```
(gv CON-CIRCLES :white-circle :top)
```

or even

```
(gv CON-CIRCLES :white-circle :parent)
```

This is the end of the section regarding constraints. Destroy the window we have been using to keep it from interfering with future examples in the tutorial.

```
(opal:destroy CONSTRAINTS-WIN)
```

5. Interactors

Interactors are used to communicate with the mouse and keyboard. Sometimes you may just want a function to be executed when the mouse is clicked, but often you will want changes to occur in the graphics depending on the actions of the mouse. Examples include moving objects around with the mouse, editing text with the mouse and keyboard, and selecting an object from a given set.

The fundamental way that the interactors communicate with graphical objects is that they set slots in the objects in response to mouse movements and keyboard key strokes. That is, they generate side effects in the objects that they operate on. For example, some interactors set the `:selected` and `:interim-selected` slots to indicate which object is currently being operated on. When objects are defined with formulas that depend on these special slots, the appearance of the objects (i.e., the graphics of the interface) can change in response to the mouse.

It is important to note that all of the gadgets come with their interactors already defined. Therefore, you do not need to create interactors that change the gadgets.

In this section we will see some examples of how to change graphics in conjunction with interactors. Section 8.5 describes how to use an important debugging function for interactors called `trace-inter`. Although this tutorial only gives examples of using the `button-interactor` and `move-grow-interactor`, the Interactors Manual discusses and provides examples for all six types of Garnet interactors.

5.1. Kinds of Interactors

The design of the interactors is based on the observation that there are only a few kinds of behaviors that are typically used in graphical user interfaces. Currently, Garnet supports seven types of interactive behavior, which allows a wide variety of user actions in an interface. Below is a list of the available interactors, which are described in detail in the Interactors Manual.

`Menu-Interactor` - For selecting one or more choices from a set of items. Useful in menus, etc., where the mouse may be held down and dragged while moving over the items to be selected.

`Button-Interactor` - For selecting one or more choices from a set of items. Useful in single buttons and panels of buttons where the mouse can only select one item per mouse click.

`Move-Grow-Interactor` - For moving and changing the size of an object. Useful in scroll bars and graphics editors.

`Two-Point-Interactor` - For obtaining one or two points in a window from the user. Useful in specifying the size and position of a new object to be created.

`Angle-Interactor` - For getting the angle that the mouse moves around a point. Useful in circular gauges or for "stirring motions".

`Text-Interactor` - For editing strings. Most useful for small strings, since Garnet does not support complicated word-processing applications.

`Gesture-Interactor` - For recognizing gestures drawn by the user (e.g., the user draws a rough shape that Garnet recognizes as a square).

`Animator-Interactor` - For executing a function at regular intervals, allowing rapid updating of graphics.

There are also several interactors that work with the `multifont-text` object. This object and its associated interactors are discussed in the Opal manual.

5.2. The Button Interactor

In this example, we will perform an elementary operation with an interactor. We will create a window with a white rectangle inside, and then create an interactor that will make it change colors when the mouse is clicked inside of it. First, create the window with the white rectangle.

```
(create-instance 'INTER-WIN inter:interactor-window
   (:left 750)(:top 80)(:width 250)(:height 250))
(create-instance 'TOP-AGG opal:aggregate)
(s-value INTER-WIN :aggregate TOP-AGG)
(opal:update INTER-WIN)

(create-instance 'CHANGING-RECT opal:rectangle
   (:left 20) (:top 30)
   (:width 60) (:height 40)
   (:filling-style (o-formula (if (gvl :selected)
                                  opal:black-fill
                                  opal:white-fill)))
   (:selected NIL))   ; Set by the interactor

(opal:add-components TOP-AGG CHANGING-RECT)
(opal:update INTER-WIN)
```

From the definition of the `:filling-style` formula, you can see that if the `:selected` slot in CHANGING-RECT were to be set to be non-NIL, then its color would turn to black. Conveniently, setting the `:selected` slot is one of the side effects of the `button-interactor`. The following code defines an interactor which will set the `:selected` slot of CHANGING-RECT, which will cause it to change colors.

```
(create-instance 'COLOR-INTER inter:button-interactor
   (:window INTER-WIN)
   (:start-where (list :in CHANGING-RECT))
   (:start-event :leftdown))

; Unless using CMU, Allegro, Lucid, LispWorks, or MCL
(inter:main-event-loop)
; Hit F1 while the mouse is in the Garnet window to exit
```

The `main-event-loop` function causes Garnet to start paying attention to events (like clicking the mouse) that trigger the interactors. (A background process in CMU, Allegro, Lucid, LispWorks, and MCL always pays attention to events.) Now you can click on the rectangle repeatedly and it will change from white to black, and back again. From this observation, and knowing how we defined the `:filling-style` of CHANGING-RECT, you can conclude that the `button-interactor` is setting (and clearing) the `:selected` slot of the object. This is one of the functions of the `button-interactor`. When you are ready to resume typing in the Lisp process, you have to hit the F1 key while the mouse is in the Garnet window to get a new prompt. (You may execute the `main-event-loop` call again at any Lisp prompt.)

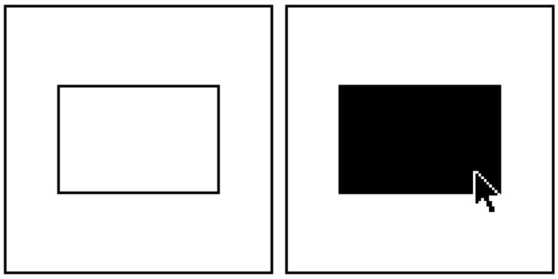

Figure 5-1: The rectangle CHANGING-RECT when its `:selected` slot is NIL (the default), and when it is set to T by the interactor (when the mouse is clicked over it).

5.3. A Feedback Object with the Button Interactor

The method we used in the previous section with the `button-interactor` involved setting the `:selected` slot of the selected object. There is another way to use the button interactor which involves using feedback objects. A *feedback object* is some object that indicates the currently selected object. For example, it is often desirable that the actual selected object not move or change color, but rather that a separate object follow the mouse or appear over the selection.

In an earlier example in section 4.4, we defined a selection box which works just like a feedback object. When the `:obj-over` slot of the selection box was set to the name of the selected object, then the box appeared around the selected object. In this example, we will redefine the objects from that section and create an interactor to work on them.

The following code is analogous to what was presented in section 4.4, but here the three selectable objects are defined as components in an aggregadget. Type in the following aggregadget and feedback object, and add them to the current window. Notice that because of the `:visible` formula in FEEDBACK-RECT, that rectangle will be invisible when the window is first updated.

```
(create-instance 'AGG opal:aggregadget
   (:top 100)
   (:parts
    `((:white-rect ,opal:rectangle
                   (:left 20)
                   (:top ,(o-formula (gvl :parent :top)))
                   (:width 60)
                   (:height 40)
                   (:filling-style ,opal:white-fill))
      (:gray-rect ,opal:rectangle
                  (:left 110)
                  (:top ,(o-formula (gvl :parent :top)))
                  (:width 60)
                  (:height 40)
                  (:filling-style ,opal:gray-fill))
      (:black-circle ,opal:circle
                     (:left 200)
                     (:top ,(o-formula (gvl :parent :top)))
                     (:width 40)
                     (:height 40)
                     (:filling-style ,opal:black-fill)))))

(create-instance 'FEEDBACK-RECT opal:rectangle
   (:obj-over NIL)   ; A pointer slot to be set by the interactor
   (:visible (o-formula (gvl :obj-over)))
   (:left (o-formula (- (gvl :obj-over :left) 10)))
   (:top (o-formula (- (gvl :obj-over :top) 10)))
   (:width (o-formula (+ 20 (gvl :obj-over :width))))
   (:height (o-formula (+ 20 (gvl :obj-over :height))))
   (:line-style opal:line-4))

(opal:add-components TOP-AGG AGG FEEDBACK-RECT)
(opal:update INTER-WIN)
```

Notice that the `:obj-over` slot of FEEDBACK-RECT is a pointer slot, as usual. When `:obj-over` is set with the name of an object, the FEEDBACK-RECT will appear over the object because of the way we defined its position and dimension formulas. In this example, we will not set `:obj-over` by hand, as we did previously. Instead, we will create a `button-interactor` to set the slot for us.

The following code defines an interactor which uses the FEEDBACK-RECT to indicate which object is selected. Since all of the selectable objects are in the same aggregate, we can tell the interactor to start whenever the mouse is clicked over any component of AGG.

```
(create-instance 'SELECTOR inter:button-interactor
   (:window INTER-WIN)
   (:start-where (o-formula (list :element-of AGG)))  ; Work on the components of AGG
   (:final-feedback-obj FEEDBACK-RECT)
   (:how-set :toggle))

; Unless using CMU, Allegro, Lucid, LispWorks, or MCL
(inter:main-event-loop)
; Hit F1 while the mouse is in the Garnet window to exit
```

Now when you click on the objects, the feedback object will appear over the selected object. The reason is that the `button-interactor` sets the `:obj-over` slot of the feedback object. Since the interactor is a toggling interactor (according to its `:how-set` slot), the `:obj-over` slot will be reset to NIL when the selected object is clicked on again.

If you entered the `main-event-loop`, remember to hit the F1 key before typing in the next example.

5.4. The Move-Grow Interactor

From the previous example, you can see that it is easy to change the graphics in the window using the mouse. We are now going to define several more objects in the window and create an interactor to move them.

The side effect of the `move-grow-interactor` is that it sets the `:box` slot of the selected object (as well as the feedback object, if any) to be a list of four values -- the left, top, width, and height of the object. When formulas are defined in the `:left`, `:top`, `:width`, and `:height` slots which depend on the `:box` slot, then the position and dimensions of the object will change whenever the `:box` slot changes.

The idea goes like this: Suppose the current value of the `:box` slot in a rectangle is '(0 0 40 40). Since the `:left` and `:top` slots are constrainted to the `:box` slot, the rectangle appears at the position (0,0). To move the object, the user clicks and drags on the rectangle until it is at position (50,50). When the user lets go, then the interactor automatically sets the `:box` slot to '(50 50 40 40). Since the `:box` slot has changed, the formulas in the `:left` and `:top` slot are re-evaluated, and the rectangle appears at the new position.

The following code creates a prototype circle and several instances of it. With a little study, it should be clear how the position and dimension formulas work with respect to the `:box` slot. All of the circles are then added to an aggregate, and this aggregate is added as a component to the top-level aggregate.

```
(create-instance 'MOVING-CIRCLE opal:circle
   (:box '(0 0 40 40))  ; This slot will be set by the interactor
   (:left (o-formula (first (gvl :box))))      ; Get the first value in the list
   (:top (o-formula (second (gvl :box))))
   (:width (o-formula (third (gvl :box))))
   (:height (o-formula (fourth (gvl :box)))))

(create-instance 'M1 MOVING-CIRCLE
   (:box '(120 30 40 40)))

(create-instance 'M2 MOVING-CIRCLE
   (:box '(30 100 60 60)))

(create-instance 'M3 MOVING-CIRCLE
   (:box '(120 100 80 80)))

(create-instance 'CIRCLE-AGG opal:aggregate)

(opal:add-components CIRCLE-AGG M1 M2 M3)

(opal:add-components TOP-AGG CIRCLE-AGG)
(opal:update INTER-WIN)
```

If you want to try setting the `:box` slot of any of these objects, you will see how the position and dimension of each circle depend on it. Be sure you set the `:box` slot to be a list of four positive numbers, or an error will occur!

Now let's create an instance of the `move-grow-interactor`, which will cause the moving circles to change position. The following interactor works on all the components of the aggregate CIRCLE-AGG. Remember to execute the `inter:main-event-loop` call to start the interactors working.

```
(create-instance 'MG-INTER inter:move-grow-interactor
   (:window INTER-WIN)
   (:start-where (list :element-of CIRCLE-AGG)))     ; Work on the components of CIRCLE-AGG

; Unless using CMU, Allegro, Lucid, LispWorks, or MCL
(inter:main-event-loop)
; Hit F1 while the mouse is in the Garnet window to exit
```

Now if you press and drag in any of the circles, they will follow the mouse. This is because the interactor sets the `:box` slot of the object that it is pressed over, and the `:left` and `:top` slots of the objects depend on the `:box` slot.

It is worth noting once again that the `move-grow-interactor` does not set the `:left`, `:top`, etc. slots of the selected object. It instead sets the `:box` slot of the selected object, and the user is required to define formulas that depend on the `:box` slot.

5.5. A Feedback Object with the Move-Grow Interactor

Now let's add a feedback object to the window that will work with the moving circles. In this case, the feedback object will appear whenever we click on and try to drag a circle. The mouse will actually drag the feedback object, and then the real circle will jump to the final position when the mouse is released.

Our feedback object will be a circle with a dashed line. The DASHED-CIRCLE object defined below will have two slots set by the interactor. The `:box` slot will be set while the mouse is held down and dragged, and the `:obj-over` slot will be set to point to the circle being dragged. Given our MOVING-CIRCLE prototype, the feedback object is easy to define.

```
(create-instance 'DASHED-CIRCLE MOVING-CIRCLE
   ; Inherit all the :left, :top, etc. formulas that depend on the :box slot.
   (:obj-over NIL)   ; Set by the interactor
   (:visible (o-formula (gvl :obj-over)))
   (:line-style opal:dashed-line))

(opal:add-components TOP-AGG DASHED-CIRCLE)
```

The `:visible` slot is set with a formula because we only want the feedback object to be visible when it is being used with the interactor. Now, we will redefine the move-grow interactor to use DASHED-CIRCLE as a feedback object. (Redefining the MG-INTER will destroy the old instance, so don't worry if a warning appears.)

```
(create-instance 'MG-INTER inter:move-grow-interactor
   (:window INTER-WIN)
   (:start-where (list :element-of CIRCLE-AGG))
   (:feedback-obj DASHED-CIRCLE))
```

Now when you move the circles with the mouse, the feedback object will follow the mouse, instead of the real circle following it directly.

Since we have finished this section on interactors, destroy the window so that it does not interfere with the next example. Type the following line.

```
(opal:destroy INTER-WIN)
```

6. Programming

6.1. Creating a Panel of Text Buttons

In this section, we will go through a comprehensive example that pulls together all the concepts that have been discussed in this tutorial. The final objective will be the panel of text buttons in Figure 6-4. We will use an aggregadget to assemble a group of objects into a single button, then use an aggrelist to make multiple copies of the text button and organize them into a list for the panel, and finally create a button interactor to manage the panel.

6.1.1. The Limitations of Aggregates

Before starting the aggregadget for this example, let's take a look at the use of an aggregate. This will help to demonstrate the usefulness of aggregadgets. First, create a window with a top-level aggregate and update it:

```
(create-instance 'BUTTON-WIN inter:interactor-window
   (:left 800)(:top 10)(:width 200)(:height 400))
(create-instance 'TOP-AGG opal:aggregate)
(s-value BUTTON-WIN :aggregate TOP-AGG)
(opal:update BUTTON-WIN)
```

The TOP-AGG aggregate is the top-level aggregate for the window. If we want any object to appear in the window, it will have to be added to TOP-AGG, or added to an aggregate further below in TOP-AGG's aggregate hierarchy. We will keep TOP-AGG as the top-level aggregate throughout this example, but we will be changing its components continually.

Now we can begin adding objects to TOP-AGG (throughout this discussion you should periodically check Figure 6-4 to see why we are creating particular objects). Let's start by assembling a button. We will first create a couple of rectangles that have a fixed position so that we get the window in Figure 6-1. Since we want the rectangles to have the same dimensions, we can make a prototype object and then create two instances with appropriate position values.

```
(create-instance 'PROTO-RECT opal:rectangle
   (:width 100) (:height 50))

(create-instance 'R1 PROTO-RECT
   (:left 40) (:top 40)
   (:filling-style opal:black-fill))

(create-instance 'R2 PROTO-RECT
    (:left 20) (:top 20)
    (:filling-style opal:gray-fill))

(opal:add-components TOP-AGG R1 R2)
(opal:update BUTTON-WIN)
```

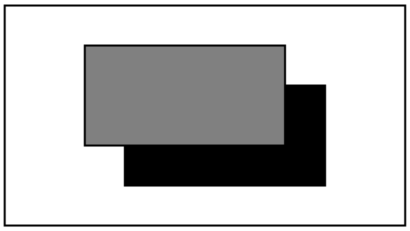

Figure 6-1: The two rectangles R1 and R2, which are instances of PROTO-RECT.

Keeping in mind our goal of making a panel of text buttons, one problem should be immediately clear. In order to make several buttons with this method, we will have to calculate the position of every rectangle in the interface and explicitly create an object for it. This will be time consuming, to say the least, and motivates us to investigate how constraints will help avoid tedious calculations. So, as the first step in pursuing a more fruitful approach, let's remove the rectangles from the window and move on to aggregadgets. To remove the rectangles, execute:

```
(opal:remove-components TOP-AGG R1 R2)
(opal:update BUTTON-WIN)
```

6.1.2. Using an Aggregadget for the Text Button

When we create an aggregadget, we essentially create an aggregate and add the components (along with pointer slots) all at once. Our task is to build an aggregadget with two rectangles as components which will look like Figure 6-1. Since we already know what we want the rectangles to look like in the window, putting a simple aggregadget together using our previously defined R1 and R2 rectangles is straightforward. However, the key to avoiding tedious calculations of the positions of our rectangles is to generalize the code. That is, we want the positions of our components to be formulas rather than absolute numbers.

For the present, let's assume that we will always be giving absolute numbers to our top-level aggregadget (but not its components). The first problem we want to address is how to devise formulas for the positions of the rectangles. By referring back to Figure 6-1, we see that the entire aggregate has its upper-left coordinate at one corner of the gray rectangle, and its lower-right coordinate on the shadow. Therefore, it is a reasonable design decision to put the left and top of the gray rectangle at the left, top corner of the aggregadget, and then put the shadow 20 pixels further below and to the right. The following code shows the definition of our new BUTTON aggregadget with formulas defined for the positions of the rectangle components.

```
(create-instance 'BUTTON opal:aggregadget
   (:left 20) (:top 20)
   (:shadow-width 100) (:shadow-height 50)
   (:parts
    `((:shadow ,opal:rectangle
            (:left ,(o-formula (+ 20 (gvl :parent :left))))
            (:top ,(o-formula (+ 20 (gvl :parent :top))))
            (:width ,(o-formula (gvl :parent :shadow-width)))
            (:height ,(o-formula (gvl :parent :shadow-height)))
            (:filling-style ,opal:black-fill))
      (:gray-rect ,opal:rectangle
               (:left ,(o-formula (gvl :parent :left)))
               (:top ,(o-formula (gvl :parent :top)))
               (:width ,(o-formula (gvl :parent :shadow-width)))
               (:height ,(o-formula (gvl :parent :shadow-height)))
               (:filling-style ,opal:gray-fill)))))

(opal:add-components TOP-AGG BUTTON)
(opal:update BUTTON-WIN)
```

After studying the BUTTON schema for a moment, several features stand out. First, there are two slots called `:shadow-width` and `:shadow-height` defined in the top-level schema, which are used by the width and height formulas of the component rectangles. The presence of these slots at the top-level will make it easier to change the appearance of the button in the future -- if we want to make it wider, we only need to change one slot, `:shadow-width`, instead of all the components' `:width` slots.

Next, it should be clear that the formulas in the `:left` and `:top` slots of the components will place the rectangles at the appropriate positions relative to each other, with the shadow further down and to the right. Finally, it is important that the shadow comes first in the order of defining the components. Objects are drawn on the screen in the order that they are added to an aggregate, so we definitely want the gray rectangle to come after the shadow.

Notice that there is no "inheritance" going on in the BUTTON aggregadget. When we want a component to get a value from its parent, we have to explicitly define a constraint that gets that value.

We have just finished the first step in creating a text button. Although there is more code in the aggregadget example than in the previous example with rectangles R1 and R2, the aggregadget code is simple and flexible. Also, almost all of the formulas that you will write in the future will resemble those in this example. The only difference will be the names of the slots and the arithmetic that is appropriate for the situation.

Now we are ready to add more objects to the button. To do this, we will not add objects to BUTTON while it is in the window. Instead, we will remove the old BUTTON aggregadget from the window and write a new one from scratch. Most of the code that we have already written will be reused, however, and if you still have a copy of the previous example on your screen, you will be able to cut-and-paste it. So execute the command that will remove the button from the top-level aggregate:

```
(opal:remove-components TOP-AGG BUTTON)
(opal:update BUTTON-WIN)
```

6.1.3. Defining Parts Using Prototypes

Before constructing an aggregadget with additional components, let’s look at another way to define components in aggregadgets. This method will make it easier for us to develop the BUTTON aggregadget by condensing some of the code and eliminating a lot of typing.

In the previous example, the components were instances of rectangles. Another way to define components is to define them as prototypes separate from the aggregadget, and then create instances of those prototypes in the aggregadget `:parts` slot. The following code comes from this alternate method for defining aggregadgets.

```
(create-instance 'SHADOW-PROTO opal:rectangle
   (:left (o-formula (+ 20 (gvl :parent :left))))
   (:top (o-formula (+ 20 (gvl :parent :top))))
   (:width (o-formula (gvl :parent :shadow-width)))
   (:height (o-formula (gvl :parent :shadow-height)))
   (:filling-style opal:black-fill))

(create-instance 'GRAY-PROTO opal:rectangle
   (:left (o-formula (gvl :parent :left)))
   (:top (o-formula (gvl :parent :top)))
   (:width (o-formula (gvl :parent :shadow-width)))
   (:height (o-formula (gvl :parent :shadow-height)))
   (:filling-style opal:gray-fill))

(create-instance 'BUTTON opal:aggregadget
   (:left 20) (:top 20)
   (:shadow-width 100) (:shadow-height 50)
   (:string "Button")
   (:parts
    `((:shadow ,SHADOW-PROTO)
      (:gray-rect ,GRAY-PROTO))))

(opal:add-components TOP-AGG BUTTON)
(opal:update BUTTON-WIN)
```

Notice that this way of looking at aggregadgets is entirely consistent with the previous aggregadget definition. In the `:parts` slot of our new button, we have created instances just like before, but we have not explicitly defined any slots in the components. It does not matter whether we set slots in the prototype objects or in the parts definitions. Using this abbreviation method for defining aggregadgets, we can now avoid retyping the slot definitions for the old components and move on to talking about new ones.

It should be noted that the SHADOW-PROTO and GRAY-PROTO rectangles can not be added to a window by themselves. If you were to try this, you would get an error when Garnet tried to evaluate any of the formulas that we defined. This is because there is no `:parent` for either the SHADOW-PROTO or the GRAY-PROTO, which is clearly needed by the formulas. But when instances of these rectangles are added to the BUTTON aggregadget, their `:parent` slots are set to be the parent aggregadget.

As usual, remember to remove the current BUTTON from the window using `remove-components`.

6.1.4. The Label of the Button

Referring to Figure 6-4 again, we see that the text button needs a white rectangle to be centered over the gray one, and text should be centered inside the white rectangle. We will want the string of the text object to be a top level slot in the aggregadget so that we can change it easily. Thus, we need to place a constraint in the text object to retrieve it (remember the text object does not "inherit" the string from its parent just because it is a component). Other than that, the addition of the new components to the BUTTON aggregadget is straightforward. Using the abbreviation method for defining aggregadgets, we get the following code (the components SHADOW-PROTO and GRAY-PROTO were defined above).

```
(create-instance 'WHITE-PROTO opal:rectangle
   (:left (o-formula (+ 7 (gvl :parent :gray-rect :left))))
   (:top (o-formula (+ 7 (gvl :parent :gray-rect :top))))
   (:width (o-formula (- (gvl :parent :gray-rect :width) 14)))
   (:height (o-formula (- (gvl :parent :gray-rect :height) 14)))
   (:filling-style opal:white-fill))

(create-instance 'TEXT-PROTO opal:text
   (:left (o-formula (+ (gvl :parent :white-rect :left)
                        (round (- (gvl :parent :white-rect :width)
                                  (gvl :width))
                               2))))
   (:top (o-formula (+ (gvl :parent :white-rect :top)
                       (round (- (gvl :parent :white-rect :height)
                                 (gvl :height))
                              2))))
   (:string (o-formula (gvl :parent :string))))

(create-instance 'BUTTON opal:aggregadget
   (:left 20) (:top 20)
   (:shadow-width 100) (:shadow-height 50)
   (:string "Button")
   (:parts
    `((:shadow ,SHADOW-PROTO)
      (:gray-rect ,GRAY-PROTO)
      (:white-rect ,WHITE-PROTO)
      (:text ,TEXT-PROTO))))

(opal:add-components TOP-AGG BUTTON)
(opal:update BUTTON-WIN)
```

Although the centering formulas for the `:left` and `:top` slots of the text object are a little more complicated, they are basic calculations that find the proper position of the text based on the dimensions of the white rectangle. Another aspect of the formulas is that they reference not only slots in the parent object, but also slots in their sibling objects. Specifically, in the `white-rect` part, the `:left` formula looks up the aggregate tree to the `:parent`, and then looks down again into the `gray-rect`. The same tracing of the aggregate tree is involved in the `text` formulas for `:left` and `:top`.

6.1.5. Instances of the Button Aggregadget

It should be clear by now that aggregadgets are particularly useful for organizing and defining components. After creating the four prototype objects by themselves, we were able to define BUTTON with the compact aggregadget above. And with our current definition of BUTTON, we will now see another significant use of aggregadgets. The following code creates several instances of the BUTTON aggregadget, which we can use as a prototype. When you add these instances to the window, you see that component rectangles and text are generated automatically for the instances.

```
(create-instance 'BUTTON-1 BUTTON
   (:left 130) (:top 80)
   (:shadow-width 60)
   (:string "abcd"))

(create-instance 'BUTTON-2 BUTTON
   (:left 10) (:top 120)
   (:string "Button-2"))

(opal:add-components TOP-AGG BUTTON-1 BUTTON-2)
(opal:update BUTTON-WIN)
```

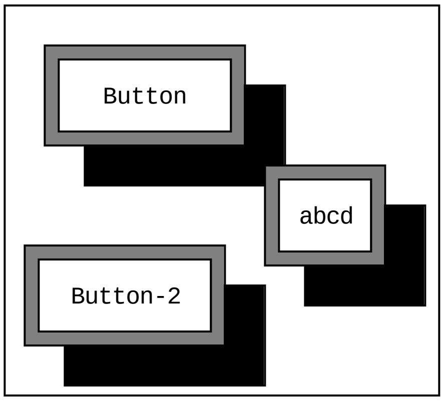

Figure 6-2: The BUTTON object and two instances of it.

This feature of aggregadgets means that you do not need to manually create objects individually and add them to the window. Instead, you can create a group of objects, and then create instances of the group.

Before moving to the next section, remember to remove your three button objects from the window with `remove-components`.

6.1.6. Making an Aggrelist of Text Buttons

Even though instances of the aggregadget will automatically generate components, the instances BUTTON-1 and BUTTON-2 show that we still need to manually supply coordinates to the aggregadget in order to position it. When we create the finished text button panel in Figure 6-4, we don't want to calculate the position of each text button in the window. (Notice that this is similar to the problem we faced several sections ago -- that we didn't want to compute the position of each rectangle within a text button.) The solution to this problem (as before) is to use a special type of aggregate that will generate components for us. This time, we will use an aggrelist.

Just for a start, let's create a simple itemized aggrelist. We will supply an item-prototype and a number to the aggrelist, and it will generate that number of instances of the item-prototype. Specifically, we want the aggrelist to make five copies of the BUTTON aggregadget. So, let's try the following code.

```
(create-instance 'PANEL opal:aggrelist
   (:left 30) (:top 10)
   (:items 5)
   (:item-prototype BUTTON))

(opal:add-components TOP-AGG PANEL)
(opal:update BUTTON-WIN)
```

By supplying the number 5 in the `:items` slot, we tell the aggrelist to make five copies of its item-prototype. And, because this is an aggrelist, all the copies of the prototype are automatically appropriate

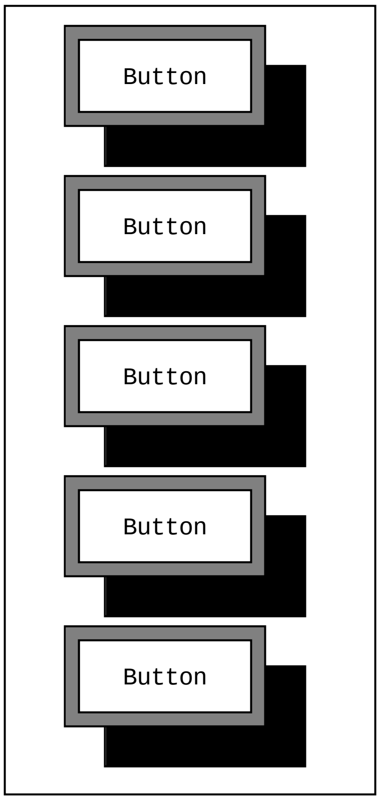

Figure 6-3: An aggrelist with an `:items` value of 5 and the BUTTON aggregadget as its `:item-prototype`.

`:left` and `:top` coordinates. It turns out that we only had to give the position of the left and top of the aggrelist; all the calculations for the buttons were handled internally. There are many customizable slots for aggrelists that change the appearance of the aggrelist -- like whether to make the panel horizontal or vertical, how much space to put between the items, etc. A list of these slots is in the Aggregadgets manual, which you can try out later.

The next step in the development of our panel is to give each button an appropriate label. To do this, we need to supply a list of labels to the aggrelist. The proper place to do this is in the `:items` slot. As we just saw, if you give the `:items` slot a number, the aggrelist generates that number of items. If instead you give it a list, then it will generate the same number of components as there are items in the list. We will also have to change the BUTTON prototype so that its `:string` slot pays attention to the list of items we supply. The following code makes this change to the BUTTON prototype in the definition of the aggrelist, so we don't have to redefine the BUTTON aggregadget.

```
(create-instance 'PANEL opal:aggrelist
   (:left 30) (:top 10)
   (:items '("Mozart" "Chopin" "Strauss" "Beethoven" "Vivaldi"))
   (:item-prototype
    '(,BUTTON
      (:string ,(o-formula (nth (gvl :rank) (gvl :parent :items)))))))

(opal:add-components TOP-AGG PANEL)
(opal:update BUTTON-WIN)
```

The `:rank` slot in the `:string` formula above is put into each component generated by the aggrelist. Even though there is no `:rank` in our definition of BUTTON, when the aggrelist generates its components, it ranks the objects in the order that they are created and stores these ranks in the `:rank` slot (ranks start at 0). This makes it easy to find the item in the `:items` list that corresponds to each component.

Since we are going to be redefining objects again, remember to remove the PANEL object from the window before going on.

6.1.7. Adding an Interactor

We are almost finished with the text button panel. At this point, the panel that we have written is still a passive graphical object -- if you press on it with the mouse, it acts just like a pile of rectangles and does nothing at all. Therefore, the next step is to add an interactor that will cause the appearance of the buttons to change whenever we click the mouse on it. Suppose we choose to use a `button-interactor` for our interface between the mouse and the panel. By applying the principles discussed in the interactor section of this tutorial, we can write the following code for our interactor.

```
(create-instance 'PRESS-PROTO inter:button-interactor
   (:window (o-formula (gvl :operates-on :window)))
   (:start-where (o-formula (list :element-of (gvl :operates-on))))
   (:start-event :leftdown))
```

The code for this interactor is short and simple. It is a prototype object just like the rectangles, but it happens to be an interactor. The `:operates-on` reference in the formulas is analogous to the `:parent` slot in objects, and the slot is automatically created when the interactor is attached to an aggregadget or aggrelist. In interactors, the `:operates-on` slot points to the aggregadget that it is attached to, just like the `:parent` slot of a graphical object points to its aggregate. Notice that we have supplied values for the two required slots in an interactor. The `:window` slot points to the window of the aggregadget that the interactor will be attached to, which is reasonable since we want the interactor to work in the same window that the graphics appear in. The value in the `:start-where` slot tells the interactor to start whenever the mouse is clicked over any component of the aggrelist (that is, over any button).

Before we attach this interactor to the PANEL aggregadget, we are going to have to change a few of the formulas in the button and its components. The question to ask is -- how should the graphics change when we press the mouse over the button? By looking at the full text-buttons picture in Figure 6-4, we see that the gray rectangle should move down and to the right, settling over the shadow. Therefore, we will have to change the formulas for the `:left` and `:top` of the gray rectangle. We do not have to change the `:left` and `:top` slots of the white rectangle or text because they are already constrained to the gray rectangle's position.

As you recall from the "Interactors" section of this tutorial, the `button-interactor` sets the `:selected` slot of the object it operates on when it is clicked on with the mouse. Additionally, the interactor will also set the `:interim-selected` slot of the object while the mouse is being held down over it. With this in mind, it would be useful to make the gray rectangle formulas depend on the `:interim-selected` slot of the aggregadget, since the gray rectangle should settle over the black rectangle when the mouse is held down over the button. A new GRAY-PROTO object that will respond to the interactor follows.

```
(create-instance 'GRAY-PROTO opal:rectangle
   (:left (o-formula (if (gvl :parent :interim-selected)
                         (gvl :parent :shadow :left)
                         (gvl :parent :left))))
   (:top (o-formula (if (gvl :parent :interim-selected)
                        (gvl :parent :shadow :top)
                        (gvl :parent :top))))
   (:width (o-formula (gvl :parent :shadow-width)))
   (:height (o-formula (gvl :parent :shadow-height)))
   (:filling-style opal:gray-fill))
```

The new formulas in the `:left` and `:top` of the new GRAY-PROTO now look at the `:interim-selected` slot of the button. When the mouse is clicked over a button, the button's `:interim-selected` slot will be set to T, causing the gray rectangle to be moved down and to the right. When the mouse is released, the `:interim-selected` slot will be set back to NIL, and the gray rectangle will return to its normal position.

If you refer back to the definitions of the other components, you will see why the gray rectangle is the only component that we had to change. The white rectangle depended on the position of the gray rectangle, and the text was centered inside the white one. Thus, when the gray rectangle's position changed, the change was propagated to all of the dependent formulas, resulting in uniform movement of the components.

There is one final note to make before we can complete the panel. When the gray rectangle is pushed down onto the shadow, the dimensions of the button are going to change. That is, when the gray rectangle covers the shadow completely, then the button's aggregate has smaller dimensions than if the two rectangles are spread out a bit. If left unchecked, this will cause unexpected behavior because the aggrelist keeps the components arranged according to their width and height. For this reason, we will have to supply our own `:width` and `:height` values to the BUTTON aggregadget within our definition of the PANEL. To see the problem for yourself, you may want to leave out the `:width` and `:height` values in the following code just to see what happens. Then you will certainly want to try the code again with the values in place.

```
(create-instance 'BUTTON opal:aggregadget
   (:left 20) (:top 20)
   (:width 120) (:height 70)   ; The dimensions of the two rectangles plus their offset
   (:shadow-width 100) (:shadow-height 50)
   (:string "Button")
   (:parts
    `((:shadow ,SHADOW-PROTO)
      (:gray-rect ,GRAY-PROTO)
      (:white-rect ,WHITE-PROTO)
      (:text ,TEXT-PROTO))))

(create-instance 'PANEL opal:aggrelist
   (:left 30) (:top 10)
   (:items '("Mozart" "Chopin" "Strauss" "Beethoven" "Vivaldi"))
   (:item-prototype
    `(,BUTTON
      (:string ,(o-formula (nth (gvl :rank) (gvl :parent :items))))))
   (:interactors
    `((:press ,PRESS-PROTO))))

(opal:add-components TOP-AGG PANEL)
(opal:update BUTTON-WIN)

; Required if you are not using CMU, Allegro, Lucid, LispWorks, or MCL
(inter:main-event-loop)   ; To start the interactors.
                          ; Hit F1 in the Garnet window to exit.
```

Once you have entered the `main-event-loop` (not necessary in CMUCL, Allegro, Lucid, LispWorks, or MCL), you can click on any of the buttons and they will respond. The movement comes from the interactor setting the `:interim-selected` slot of the button that you press on, which causes the `:left` and `:top` slots of the components to be re-evaluated. When you let go, the `:interim-selected` slot is cleared, and the components return to their original position.

Remember to destroy the window when you are finished with this example.

```
(opal:destroy BUTTON-WIN)
```

6.2. Referencing Objects in Functions

If a function that returns an object is only going to be called once, then it is usually appropriate to explicitly name the objects in each `create-instance` call. This is the method used in the demonstration functions that accompany the gadgets. However, if a function is called repeatedly and returns objects which will be used at the same time, then unnamed objects should probably be used.

For example, the function below will create and return windows with messages in them. If the window in

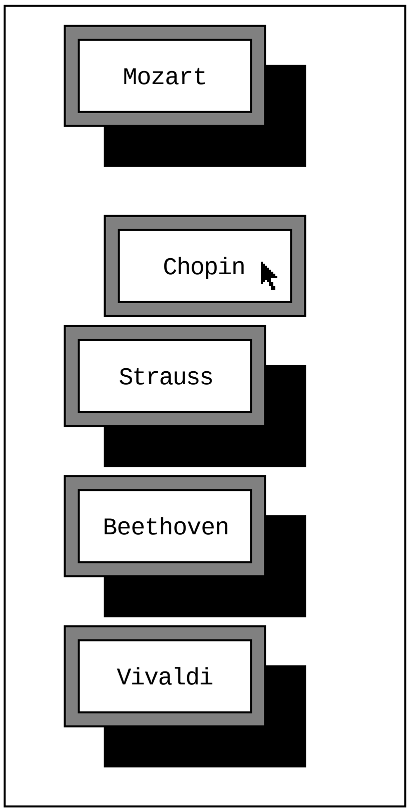

Figure 6-4: The finished text button panel.

the function was explicitly named (say 'WIN or something), then each call to the function would destroy the previous window instance while creating the new one.

```
(defun Make-Win (left top string)
  ; Create unnamed objects and assign them to local variables
  (let ((win (create-instance NIL inter:interactor-window
                (:left left) (:top top)
                (:width 100) (:height 100)))
        (agg (create-instance NIL opal:aggregate))
        (message (create-instance NIL opal:text
                    (:left 20) (:top 40)
                    (:string string))))
    ; Manipulate the objects according to their local names
    (s-value win :aggregate agg)
    (opal:add-component agg message)
    (opal:update win)
    win))   ; Return the internal name of the new window

(setf Win-List (list (Make-Win 100 100 "Hello")
                     (Make-Win 190 120 "Window 2")
                     (Make-Win 70 190 "Third Win")))
```

Each time `Make-Win` is called, a window is created, an aggregate is attached, and a text object is added to the aggregate. We kept a list of the internal names of the windows in `Win-List` because we will want to destroy them later. Each of the windows in the list can be manipulated as usual (using `s-value`, etc.) by referring to their generated names.

To clean up the windows generated from `Make-Win`, you could use `dolist` to destroy the whole list, or manually destroy the windows individually while referring to their generated names.

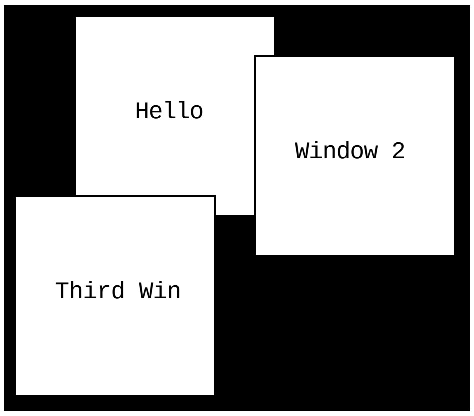

Figure 6-5: Three windows created with the function `Make-Win`.

7. Hints and Caveats

There is a small manual devoted to optimizations that can be added to your Garnet programs that make them smaller and faster. This section lists a few suggestions that are sometimes required by Garnet programs, in addition to helping your programs be more efficient.

7.1. Dimensions of Aggregates

7.1.1. Supply Your Own Formulas to Improve Performance

Although it is usually not necessary to specify the `:width` and `:height` of an aggregate, the programmer can almost always define formulas that will be more efficient than the default formulas for computing the bounding box. The default formulas look at all the components of the aggregate and compute the appropriate bounding box, but they are completely generic and make no assumptions about the arrangement of the components. Since you will know where the components will appear on the screen, you can usually supply simple formulas that depend on only a few of the components.

For example, if you created an aggregadget out of nested rectangles, where there was one outer rectangle and several others inside of it, then you would want to define dimension formulas for the aggregate that depend only on the outer rectangle and ignore the inner ones. Otherwise, the default formulas would check every rectangle before deciding on the correct width and height of the aggregate.

7.1.2. Ignore Feedback Objects in Dimension Formulas

A good reason to define your own formulas for the `:width` and `:height` slots of aggregates is that you usually don't want the feedback object to be considered in the bounding box calculation.

7.1.3. Include All Components in the Aggregate's Bounding Box

Components of aggregates should always be inside the bounding boxes of the aggregates. That is, you should not make the `:left` of an aggregate be 40, and then the left of a component be (- 40 offset). This will put the component outside of the bounding box of the aggregate (too far to the left), and Garnet will not be able to update the aggregate properly.

The solution here is to make the left of the aggregate be the same as the leftmost component, and then make the component inherit that left. Of course, if you have several components which all have different lefts, then you will have to add offsets to the lefts of the other components.

7.2. Dimensions of Windows

Don't make the size of windows depend on the size of the objects inside. This will lead to frequent refreshing of the entire window, causing very poor performance.

7.3. Formulas

7.3.1. The Difference Between formula and o-formula

The difference between `formula` and `o-formula` is somewhat confusing. The preferred form is `(o-formula (expression))` because the expression will be compiled when the the file is compiled. Then, at run-time, the expression for the constraint executes as compiled code when the formula needs to be re-evaluated. (This works by expanding the code in-line to create a lambda expression, for which the compiler generates code.) When the `(formula '(expression))` form is used, the expression is interpreted at run-time, so the constraint executes much slower.

The disadvantage of `o-formula`, however, is that because it is a macro, variable references do not create lexical closures. This means that variables referenced inside an `o-formula` will not be expanded into their actual value inside the expression. The variable name will instead remain inside the expression, and if its value ever changes, the new value will be reflected when the expression is reevaluated.

On the other hand, using the form `(formula `(... ,*variable* ...))` puts the value of `*variable*` permanently in the formula, and eliminates the reference to `*variable*`. If all your object references use `(gvl ...)` to get values out of slots of the object ("paths" in aggregadgets), then this is not relevant, and you should use `o-formula`.

As an example, let's start with a global variable and two formulas that use the variable. One formula will be an `o-formula`, and one will be a plain `formula`. Note: `lisp>` represents the prompt for the lisp listener.

```
lisp> (setf *width* 100)
100
lisp> (create-schema 'A
         (:left 10)
         (:right1 (o-formula (+ (gvl :left) *width*)))
         (:right2 (formula `(+ (gvl :left) ,*width*))))
Object A

#k<A>
lisp> (gv A :right1)
110
lisp> (gv A :right2)
110
lisp>
```

So in both cases the formula computed the sum of the left and the current value of `*width*`. Now, what happens if we change `*width*`? At first, it seems that nothing happens. Just changing the value of the variable will not cause the formulas to recompute -- only things that are `gv`'ed have dependencies, and Garnet doesn't know that the variable's value has changed yet.

```
lisp> (setf *width* 22)
22
lisp> (gv A :right1)
110
lisp> (gv A :right2)
110
lisp>
```

But now let's change the value of the `:left` slot, which will invalidate both formulas and will cause them to recompute.

```
lisp> (s-value A :left 33)
33
lisp> (gv A :right1)
55
lisp> (gv A :right2)
133
lisp>
```

Now they recomputed, and the difference is obvious. In the `o-formula`, the `*width*` variable reference was still hanging around, and that expression used the current value of `*width*`. A `ps` of the `o-formula` shows it's still there:

```
lisp> (ps (get-value A :right1))
{#k<F74>
  lambda:        (+ (gvl :left) *width*)
  cached value:  (55 . T)
  on schema #k<A>, slot :RIGHT1
  }

NIL
lisp>
```

On the other hand, the plain formula got rid of the `*width*` variable when the "," dereferenced it.

```
lisp> (ps (get-value A :right2))
{#k<F73>
  lambda:        (+ (gvl :left) 100)
  cached value:  (133 . T)
  on schema #k<A>, slot :RIGHT2
  }

NIL
lisp>
```

Notice the 100 replaced `*width*` in the definition of the formula.

One occasion where this distinction between `formula` and `o-formula` comes up is the creation of objects while iterating over a list. The following code correctly dereferences the variable `obj` when constructing formulas.

```
(dolist (obj objlist)
  (create-instance NIL opal:rectangle
     (:left (formula `(gv ,obj :left)))
  ))
```

7.3.2. Avoid Real Number Divide

In all graphical objects, the position and dimension slots `:left`, `:top`, `:width`, and `:height` all take integer values. Therefore, the integer divide functions `round`, `floor`, and `ceiling`, etc. should be used more frequently than `/` for division.

7.4. Feedback Objects

If all of the components of an aggregate are selectable, then any feedback object should be put in a separate aggregate so that the feedback object itself is not selectable.

8. Debugging

The Debugging Tools Reference Manual documents many functions that are useful in answering the most common questions that users have when developing Garnet code. The functions will help you find objects, explain the values of particular slots, describe inheritance and aggregate hierarchies, and inspect constraints and interactors. This section describes the most commonly used functions for examining Garnet objects.

8.1. The Inspector

There is a powerful debugging tool called the `Inspector` which allows you to examine and change slot values of your objects without typing into the lisp listener. This tool can be invoked by hitting the HELP key while the mouse is positioned above the object to be examined.

You can easily try this tool if you have any Garnet window with objects in it. Sections 2.2-2.4 of this tutorial provide a simple example window with step-by-step interaction with the Inspector.

8.2. PS

`kr:PS` *object*

The function `ps` (which stands for "print schema") is used to examine individual schemas. When `ps` is called with a Garnet object, a list of all the object's slots and values will be printed. By default, any slot whose value is inherited from a prototype is not printed unless `gv` has already been called on that slot.

The `ps` function resides in the KR package, and is fully documented in the KR manual. There are several switches and global variables that control the amount of information that `ps` prints.

8.3. Flash

`garnet-debug:Flash` *object* `&optional` *n*

The function `flash` helps you to visually locate *object* in a window by flashing the bounding box of *object* from black to white *n* times. The *object* must already be in a window in order for it to flash. If `flash` is unable to flash the object, then the function will try to give you some explanation of why the object will not flash.

8.4. Ident

`garnet-debug:Ident`

The function `ident` takes no parameters. After you call `ident`, Garnet waits for the next input event in any Garnet window, like clicking the mouse. If you click over an object, then the name of the object will be printed along with some other information about the object and the window.

Clearly, this function is useful if there are many objects in a window and you forget the names of all of them. A more interesting application is when there are unnamed objects in the window (that is, they were given NIL for a name in their schema definition and now have only internal names) and you want to analyze or manipulate an unnamed object. Then, `ident` will return the internal name of the object clicked on, and that name can be used in `gv` or `s-value` calls as usual.

8.5. Trace-Inter

`inter:Trace-Inter &optional` *interactor* [*Function*]
`inter:Untrace-Inter` [*Function*]

The function `trace-inter` is often used to find out why an interactor is not working as you expected. Interactor problems most often arise from improper definitions of either the interactors or the objects they work on. Using `trace-inter` can help to narrow the reasons for the unexpected behavior.

Executing `untrace-inter` will turn off the tracing for interactors.

Table of Contents