Chapter 3
Your First Program: Joe the Box

1 Adele Goldberg's Joe the Box

Joe the Box was one of the very first curricular elements for teaching Smalltalk. It was originally developed by Adele Goldberg for Smalltalk-72. Most recently, she has created a version of it for her LearningWorks Smalltalk-based environment for learning systems. Alan Kay used Joe the Box in his 1977 *Scientific American* article.

Joe the Box works as an introduction to object-oriented programming at two levels. At the first level, it's a *microworld* for exploring objects. There are a couple of different kinds of boxes, several basic operations that you can do with them, and many ways of combining the operations to do something interesting. "Microworlds" were invented by the MIT Logo Group as a way of providing a programmable space for exploring material.

At the second level, Joe the Box is an interesting small program to build. It's an interesting exploration of user input, computer graphics, and creating an interface for programmers.

Working through the second level with the book is easy: Start with a bare Squeak image, type things in, and try them! However, this is in some conflict with the first objective, which is to play with the space first. What you might do is to load up the Boxes microworld to play with it, and then delete it before exploring how it's implemented.

We'll start out by talking about filing things in as an important skill in its own right. The Smalltalk community has always been one whose members share what they do. You use what others have shared by filing them in.

2 Generating: Filing In New Code

The process of gathering someone else's code into your image (and compiling it) is called *filing in*. You usually do it from the file list. You typically file in things whose file suffix is *.st* (for SmallTalk code) or *.cs* (for Change Set, which we'll talk more about in a later chapter.)

You load up the Boxes microworld by *filing in* the source code from the disk. To file in a piece of code is to (a) load the class and method definitions into your image (i.e., compile them) and (b) execute commands to set up objects and do initializations. The source code file for Boxes is named **Boxes.st**. It's provided on the CD.

It's easy to create source files by *filing out* sections of code. FileOut files are just plain text files with exclamation points "!" in them to delimit sections. The easiest way to create one is to use the yellow-button menu over any pane in the System Browser and choose *File Out.* You can file out a whole category of classes, or any class, or any protocol of methods, or any single method. In the next chapter, we'll talk about more advanced mechanisms for managing code that appears in different categories and classes.

You find files and file them in with the File List (Figure 1). Choose *Open…* from the Desktop Menu, then *File List.* You navigate into a directory by clicking on its name on the right pane. You move up by clicking on the directory in the left pane. You can limit the filenames that appear by changing the filename pattern and then choosing Accept (yellow button menu) on the new pattern. ***** matches everything, ***.st** only matches things that end in ".st", and so on. When the file appears that you want, use the yellow-button menu above the file name and choose *File in.* After a moment (barring syntax errors), the code is loaded into your image for you to use.

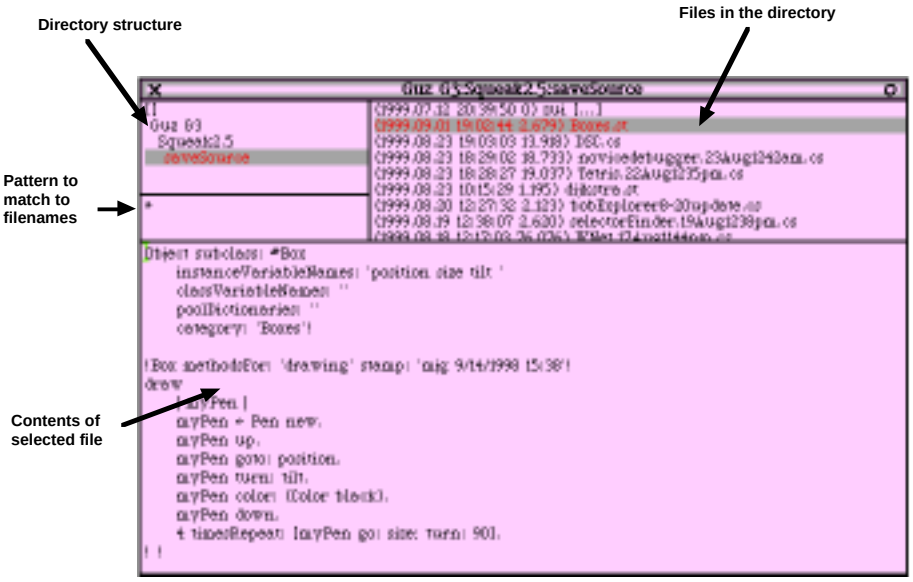

Figure 1: File List with its Pieces Identified

To remove the Boxes microworld, find the Boxes category in the System Browser. Use the yellow button menu in the category pane to remove it. It's okay *not* to remove it, too. Later sections of the chapter will show how to extend the existing code to do interesting things.

3 Playing with Boxes

First, we'll create a Box world. Create a workspace and drag it to the left of your screen, about halfway down the display. Type **Box clearWorld** and DoIt.

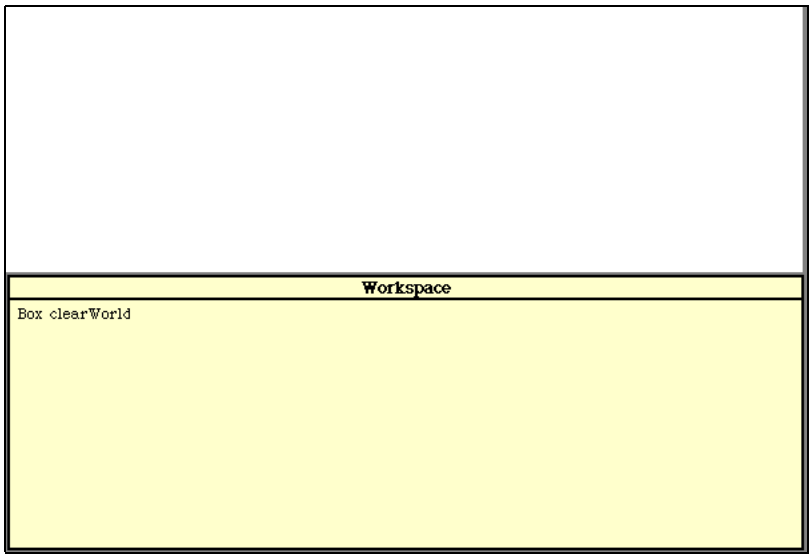

Figure 2: Creating the Box world to play in

The word **Box** does refer to an object. It's an object that defines other objects, like the master documents in Sketchpad. Box is a kind of object callled a *class*. We can create a box named Joe by asking **Box** to give us a new object.

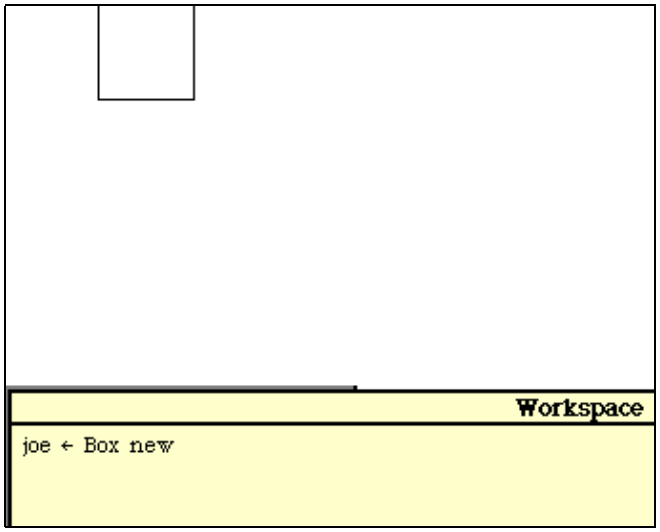

Figure 3: Creating the Box, Joe

joe is an object. **joe** is an *instance* of the class **Box**. **joe** knows that he is an instance of Box. If we ask him his class, he will print it for us. Do a PrintIt on **joe class printString**.

```
joe class printString
      'Box'
```

Figure 4: Joe knows his Class

Joe (we'll anthropormophize the object here) understands several *messages*. He knows how to turn himself a certain number of degrees.

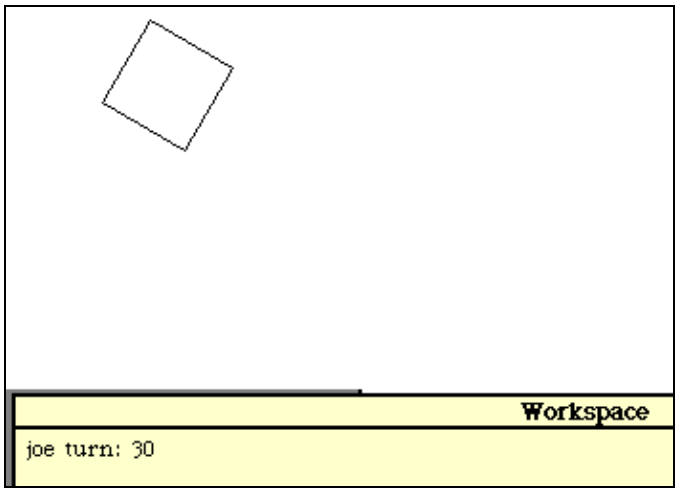

Figure 5: Joe can turn

He knows how to move himself a given number of pixels. In the message below, Joe is asked move 30 pixels horizontally (to the right) and 30 pixels vertically (down). By saying **30 @ 30**, a **Point** object is defined which is added to Joe's current location.

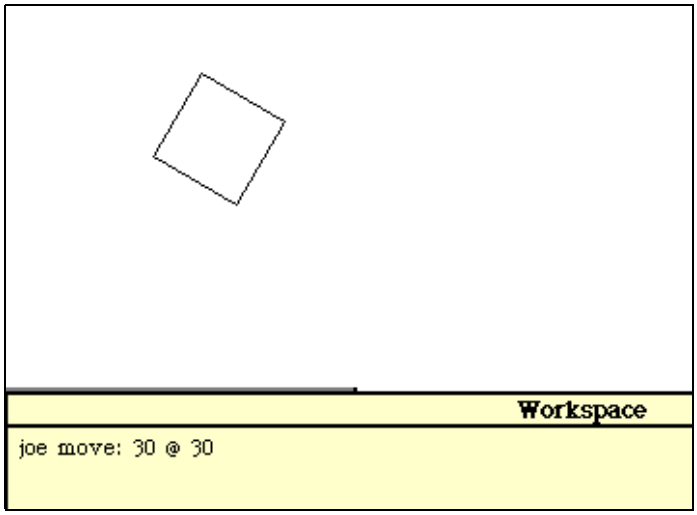

Figure 6: Joe can move

He also knows how to make himself grow larger or smaller by a certain number of pixels.

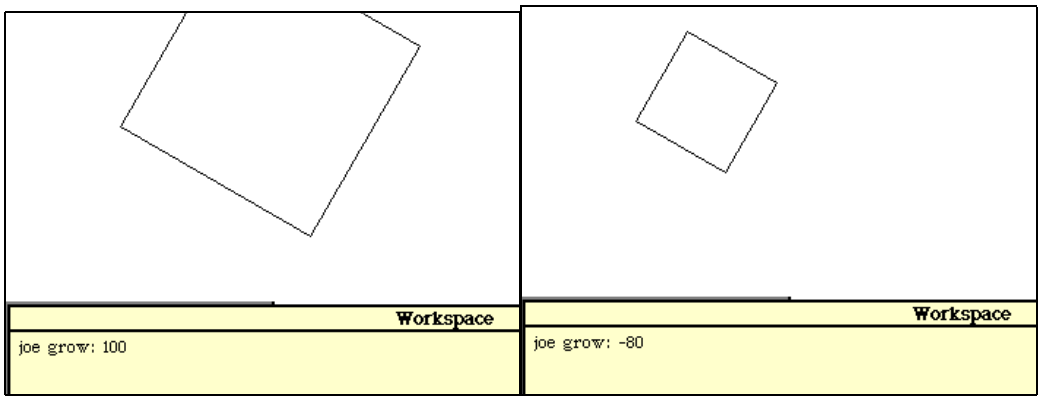

Figure 7: Making Joe grow larger and smaller

Joe also knows how to go to a given point. This message can be combined with messages understood by **Sensor** (the object that represents the hardware mouse on the device) to create an interactive Joe.

```
[Sensor anyButtonPressed] whileFalse:
        [joe moveTo: (Sensor mousePoint)].
```

When this code is executed, Joe moves to whatever point the **Sensor** says that the mouse is pointing at. It keeps doing this, until a mouse button is pressed. What you see when this code is executing is that Joe follows the mouse pointer wherever it is dragged on the screen.

Let's create another Box, named Jill.

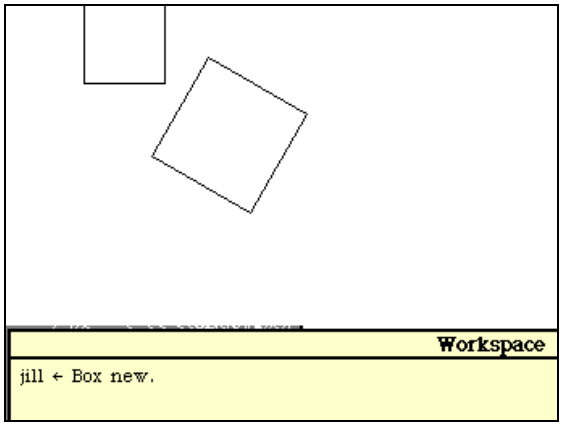

Figure 8: Creating a second Box

Jill understands all of the same messages that Joe does.

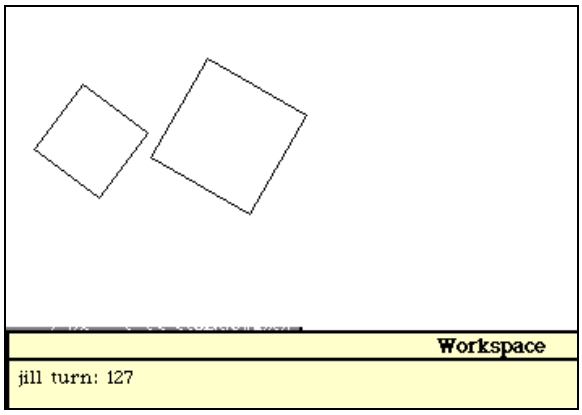

Figure 9: Jill understands Joe's messages

But Jill and Joe are complete separate objects. Joe and Jill are at separate positions on the screen, and they can have different turn angles. They cannot *directly* influence one another. Joe cannot change any aspect of Jill, nor can Jill change any aspect of Joe. All that they can do is to send messages to one another.

3.1 Joe and Jill as example objects

But Jill and Joe are both *instances* of the same *class*. They are both instances of **Box**. Classes perform several important roles, which were presented in earlier chapters. The Box world makes the roles more concrete.

- Classes group definitions of *attributes* (the data that objects carry with them) and *services* (the behavior that objects do in response to messages). Without classes, each object would have to be taught its own attributes and services. Instead, Joe and Jill both have the same attributes (e.g., an angle of rotation, a size) and the same services (e.g., **grow**:, **move**:)
- Classes provide pieces to reuse. Whole classes can be reused (e.g., maybe you want to use Boxes in another project), or you can *inherit* the attributes and services of one class into another. By creating a *subclass*, you create something which is just like the class (we say it's a *specialization* of the *generalization*, or the subclass *IsA* superclass) but can add its own special attributes and services in it, too.
- Classes act as *factories*. They produce new instances of themselves. They can also reprogram instances already created. If the definition of a given service (that is, a *method*) is changed in the class, all instances will now use the new definition of the given service. Later in the chapter, we'll redefine a method in Boxes, and both Joe and Jill will respond differently to that method

We've seen that the class **Box** understands a couple of different messages.

Class message	Meaning
new	Creates a new **Box** and has it display itself.
clearWorld	Clears a portion of the screen for Boxes.

Box *instances* understand *different* messages. **joe new** would generate an error. Instances of **Box** know the *instance* methods defined in **Box**.

Joe and Jill respond to several other messages. The table below lists the messages that **Box** provides for Joe and Jill.

Instance message	Meaning
draw	Draws the **Box** instance.
undraw	Erases the **Box** instance from the display (but it still exists).
move:	Moves a given increment, where the increment is expressed as a **Point** object.
moveTo:	Moves the instance to a specific point.
grow:	Expands or shrinks the instance as the size is specified.
turn:	Tilts the box a certain amount.

If the definition of what it means to **draw**, for example, were changed in **Box**, it would be changed for both Joe and Jill. The definition of how to behave in response to a message resides in the class. That means that all instances of the same class behave the same in response to the same messages. But the data in each object is its own—it has its own copy.

3.2 Adding a New Kind of Box

There is another *kind* of **Box**, called a **NamedBox**. Instances of **NamedBox** are **Box**es, but they have some changed features. We say that **NamedBox** is a *subclass* of **Box**. Instances of **NamedBox** know everything and can do anything that a **Box** can do, but they may know other things or respond to messages slightly differently. This is an example of *inheritance*.

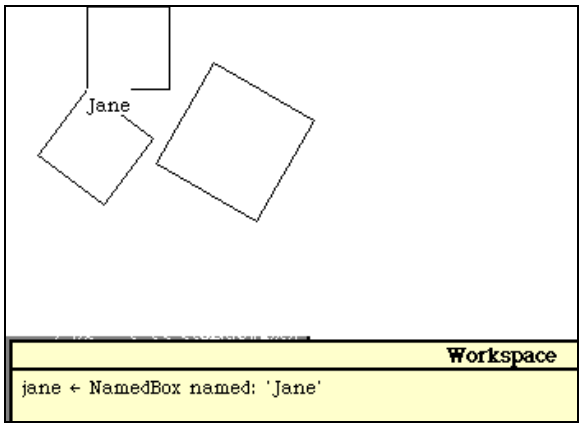

Figure 10: Creating a NamedBox

Notice that Jane draws herself differently on the screen. Jane knows that she's a kind of Box.

```
jane isKindOf: Box “PrintIt to see true”
```

And Jane knows how to do the kinds of things that Joe and Jill do.

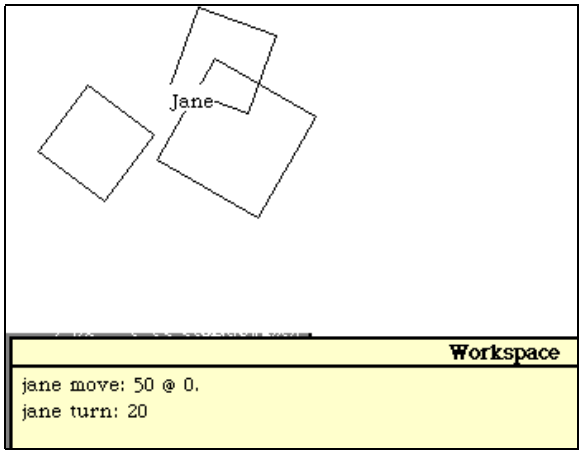

Figure 11: Jane responding to the same Joe and Jill messages

Jane, as an instance of NamedBox, really understands two messages differently than Box, and it adds the definition of one new message. The re-definition is called *overriding* the definition in the superclass.

NamedBox instance message	Meaning
draw	A NamedBox draws itself with its name.
drawNameColor:	A NamedBox knows how to draw itself in a given color (black for display, white to erase).

undraw	When a NamedBox erases itself, it also erases its name.

The rest of what an instance of **NamedBox** knows how to do is *inherited* from **Box**. The class object **NamedBox** itself does know a message that the **Box** object does not.

NamedBox class message	Meaning
named:	Creates an instance with the given name.

4 Creating the Box Class and Box Instances

The definition of the class **Box** says that instances know three things about themselves: Their position, their size, and their tilt. Every object of class Box (and any of its subclasses, such as **NamedBox**) will have these attributes, that is, these same *instance variables.* We can create the definition of Box the same way that we created **Muppet** in the previous chapter. We fill out the template and accept. If you already have the Boxes microworld loaded, you can just select the **Box** class to see the below definition.

```
Object subclass: #Box
        instanceVariableNames: 'position size tilt '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Boxes'
```

Boxes get created by sending **new** to the class. There is a class method **new** defined in Boxes. You can find (or create) this method by clicking on the class button in the browser with the class **Box** selected.

```
new
        ^super new initialize
```

This is the method that gets executed when you execute **Box new**. It says that it returns (^) whatever the superclass (**super**) of **Box** returns when sent **new**, after that new object has been initialized. The superclass of **Box** according to the class definition (above) is **Object**. To help see how this code works, the below method is equivalent:

```
new
        | temporaryNewObject |
        temporaryNewObject ← super new. “Create the new object”
        temporaryNewObject initialize.    “Initialize it”
        ^temporaryNewObject    “Return it”
```

An interesting puzzle is why the new object returned by **new** is actually an instance of the class **Box**. We know that it is because we asked Joe what his class was. Why isn't the new object an instance of Object?

What happens is that the method which actually creates new objects creates them as instances of **self**, that is, the object that received the original message. **Box new** is a message to the class **Box**, so it's **self**, and so an instance of **Box** will be created. So, whatever object is originally sent the message **new** will be the class from which the new instance will be created.

The **initialize** message is one that instances of Box understand. There is no method **initialize** for the class **Box**. Here is the instance method that actually gets executed when a new **Box** instance is created.

```
initialize
    position := 50@50.
    size := 50.
    tilt := 0.
    self draw.
```

The purpose for an initialize method in any class is to set the instance variables to the correct initial values for an object of this type. The **initialize** method for Boxes sets a default position for a new box, a default size, and a default tilt. It then asks the new **Box** to draw itself for the first time.

Before we see the **draw** method for **Box** instances, let's see what the instances are going to be drawing on. If you recall, the first statement that we executed when working with Joe the Box wasn't **joe := Box new** but **Box clearWorld**. **clearWorld** is another *class* message that **Box** understands.

```
clearWorld
    (Form extent: 600@200) fillWhite display
```

This method creates the white rectangle on which the boxes appear. Let's dissect that single line of code a bit:

- A **Form** is the Squeak object that represents a bitmap graphic. Smalltalk has always supported interesting operations on a **Form**, and Squeak extends that with support for multiple resolutions, graphics formats, and new color transformations.

- **Form extent: 600@200** creates a blank rectangular bitmap that is 600 pixels across and 200 pixels high.
- **fillWhite** makes the new **Form** instance completely white.
- **display** puts the new **Form** instance onto the computer display. The display itself is a kind of **Form** which can be accessed via the global variable **Display**. There are many options for displaying forms. **display** simply puts the form at the upper left corner of the **Display** (**0@0**). **displayAt:** displays the form at a given point, and there are many others (under the class **DisplayObject**) that can display under various transformations.

clearWorld basically paints some white on the screen as a nice backdrop to our boxes. Note that this backdrop is not a window. If you chose *restore display* from the Desktop Menu, the white form (and all our boxes) would disappear because these forms aren't on the list of objects to refresh when the screen is repainted. Nonetheless, this works as a simple place for displaying boxes.

5 Basics of Drawing

Forms are very important in Squeak, so it's worthwhile taking a short sidetrip to talk about some of the things that Squeak can do with forms.

5.1 Creating Forms

There are many ways to create a **Form**.

- The easiest way to create a **Form** is to simply grab one from the screen. **Form fromUser** will let you drag a rectangle over a section of the current display, and return it as a **Form** instance. Try **Form fromUser display** as a simple test for selecting a chunk of the display and putting it up in the upper left corner of the display. You can also assign a **Form** to a variable to keep it around for awhile.
- The second easiest is to use one of the several editors built into Squeak. Try this in MVC (but not in Morphic!):

```
| f |
f := Form fromUser.
f edit.
```

- Morphic has a wonderful editor built into it. Morphic is the alternative user interface world in Squeak mentioned in the previous chapter. From the Desktop Menu, choose *Open…*, then *New project (morphic)*. Enter the Morphic project by clicking into it.

  Click anywhere to get the World menu, then choose *new morph...* and then *make new drawing*. Draw using the various tools, then choose the button *keep*. The new sketch (actually an instance of the class

SketchMorph) can now be dragged around, or captured from the screen. You can open up an inspector on the object to do things with it. Morphic objects are manipulated via a set of colored halos (Figure 12). To bring up the menus, select the object with command-click on a Mac and alt-click on Windows. Hold your mouse over each halo for a moment to get help on what the halo does. Once you bring up an inspector on the sketch (via the *Debug* menu, on the white halo), the message **form** to a **SketchMorph** returns its form. (To do something interesting, try something like **self form display** or even **GlobalVariable := self form display**.)

Figure 12: Colored halos on a sketch

- If you have a file somewhere in GIF, JPEG, or BMP formats, you can read it into a **Form**. **Form fromFileNamed: 'my.gif'** will read in any of the above formats and will return the Form for you to manipulate. (For example, in a workspace, do **myForm := Form fromFileNamed: 'my.gif'** and then **myForm display**.)
- You can also grab images straight off the Web. Take a look at the class methods for **HTTPSocket**. For example, you can grab a GIF image from Georgia Tech's College of Computing website:

```
HTTPSocket httpGif: 'www.cc.gatech.edu/gvu/images/headers/titles/gvu-
center.gif'
```

Once you have the form, you can do amazing things with it. Squeak has a wide variety of graphics primitives available. For example, any **Form** can be shrunk by program. Try this (which is the class method **exampleShrink** in **Form**):

```
| f s |
f := Form fromUser.
s := f shrink: f boundingBox by: 2 @ 5.
s displayOn: Display at: Sensor waitButton
```

This code will let you select a section of the screen, and then will wait for you to click somewhere. It will then shrink the selection down and display it. (Note: In Morphic, try clicking on the workspace itself for dropping the image. If you try dropping the image on the desktop, the World Menu will be brought up and will overdraw your image.) Try the same example with **s := f magnify: f boundingBox by: 5 @ 5.** and you'll magnify instead of shrinking.

Besides shrinking and magnifying, you can rotate. All forms understand the message **rotateBy: someDegrees**. Here's an example that's built into Squeak:

```
| a f |
f := Form fromDisplay: (0@0 extent: 200@200).  "Save the screen"
a := 0.                                          "Rotation value"
[Sensor anyButtonPressed] whileFalse:    "Rotate until mousebutton"

        "Grab screen from mousepoint, rotate, and display"
        [((Form fromDisplay: (Sensor cursorPoint extent: 130@66))
                rotateBy: (a := a+5)) display].

f display "Put the original corner of the screen back"
```

Until you press a mouse button, this will capture a chunk 130 pixels by 66 pixels from wherever your cursor is, and rotate it in the space 200 x 200 pixels in the upper left corner of the display. The use of **a := a + 5** is an example of doing an assignment while taking the value of the assignment (the new value of **a**) as an argument in a message. The effect is to rotate the form around and around in 5 degree increments.

The lower level code that enables all of these form capabilities is **BitBlt**, the Bit Block Transfer. Basically, graphics manipulations such as these are carefully constructed memory moves: transfers of blocks of bits in specialized ways. **BitBlt** (which is actually a class in Smalltalk) also allows you to do things like create transparent sections of images, merging sections of images, and cropping images, besides the translations and rotations we've seen here. Squeak includes a new kind of **BitBlt** called **WarpBlt** that can do very powerful color manipulations. Investigate the example methods in the **WarpBlt** class to see some demonstrations.

5.2 Teaching Boxes to Draw

Now, let's see how Box instances draw themselves.

```
draw
        | myPen |
        myPen := Pen new.
        myPen up.
        myPen goto: position.
        myPen turn: tilt.
```

```
myPen color: (Color black).
myPen down.
4 timesRepeat: [myPen go: size; turn: 90].
```

Boxes draw themselves using a **Pen** class. **Pen** instances, by default, draw directly on the display, but you can get them to draw on a given **Form** by creating them with the **newOnForm:** class method.

Pens are a form of Logo turtles, which was the computational object (in our sense of the word "object") that can be used to draw. Think of a Pen as a turtle that is carrying an ink pen. It starts out facing straight up (**north**) and has the pen pressed to the surface of the paper (in our case, the display screen). You can then make drawings by sending messages to the **Pen** instance.

Pen instance message	Meaning
up	Picks the pen up, so that the turtle/pen moves without drawing
down	Puts the pen back down again for drawing
go:	Move the pen forward along its current heading so many steps (pixels on the display)
turn:	Turn the pen a given degree angle
color:	Sets the color of the pen
north	Sets the turtle's heading toward the top of the display.

The **draw** method for a **Box** just creates the **Pen**, moves it to the position of the **Box**, sets the tilt appropriately, and draws the box. **4 timesRepeat: [myPen go: size; turn: 90].** is the classic way of drawing a Box with a **Pen** or turtle.

Note that the **Box** draws itself in black. (**Color** is a class, and the class knows several messages for creating colors of various kinds.) To erase itself, **undraw**, the Box redraws itself in white.

undraw

```
| myPen |
myPen := Pen new.
myPen up.
myPen goto: position.
myPen turn: tilt.
```

```
myPen color: (Color white).
myPen down.
4 timesRepeat: [myPen go: size; turn: 90].
```

draw and **undraw** are really the heart of the **Box** class. Once we have these, all of the other messages are just:

- Undrawing the current Box representation,
- Changing one of the Box instance variables, and
- Drawing the new Box representation.

Given that basic template, here are all of the other **Box** drawing methods.

```
grow: increment
    self undraw.
    size := size + increment.
    self draw.
move: pointIncrement
    self undraw.
    position := position + pointIncrement.
    self draw.
moveTo: aPoint
    self undraw.
    position := aPoint.
    self draw.
turn: degrees
    self undraw.
    tilt := tilt + degrees.
    self draw.
```

5.3 Getting Input from the User

When first playing with Joe the Box earlier in this chapter (page 6), Joe followed the mouse through a small example in the workspace.

```
[Sensor anyButtonPressed] whileFalse:
        [joe moveTo: (Sensor mousePoint)]
```

Sensor is actually a global variable referencing an instance of **InputSensor**. The **Sensor** is the access point for the mouse and keyboard in Squeak. You can use `Sensor` to get low-level user input.

Sensor messages for accessing the keyboard	Meaning
keyboard	Return the next character that the user types
keyboardPeek	Looks at the next character that the user types and returns it, but leaves it available to be retrieved with **keyboard**.
keyboardPressed	Just returns *whether* any key has been pressed since the last time it was checked.
shiftPressed, commandKeyPressed, controlKeyPressed, macOptionKeyPressed	Indicates whether the specified modifier key is currently pressed.

Sensor messages for accessing the mouse	Meaning
mousePoint, cursorPoint	Point where the mouse/cursor currently is
anyButtonPressed, noButtonPressed, redButtonPressed, yellowButtonPressed, blueButtonPressed	Returns true if any button is pressed (**anyButtonPressed**), no button is pressed, or if the specified button is pressed.
waitButton, waitClickButton, waitNoButton	Pauses execution until some mouse button is pressed, or until it's both pressed and released (a *click*), or until all buttons are released.

6 Extending Box to create NamedBox

The **NamedBox** is a subclass of **Box** which means that it *inherits* all of attributes (instance variable definitions) and services (methods) that **Box** already has. But the **NamedBox** can specialize and extend those definitions.

Box subclass: #NamedBox

```
instanceVariableNames: 'name '
classVariableNames: ''
poolDictionaries: ''
category: 'Boxes'!
```

In the definition of **NamedBox**, we see that one new instance variable has been added, `name`. So, **NamedBox** instances have the same **position**, **tilt**, and **size** instance variables that **Box** instances have, but **NamedBox** instances also have a **name**. None of the existing **Box** methods manipulate the name instance variables, so we need to create *accessors* if we want to be able to access the name from the outside of this object.

```
name
    ^name

name: aName
    name := aName
```

Smalltalk can differentiate between **name** and **name:**. We could have named these **getName** and **setName:**, for getting and setting the value of the `name` instance variable. Get/set methods are the style in Java. In Smalltalk style, the instance variable name without the colon is typically the getter, and the instance variable name with the colon is the setter.

We might now try to create a **NamedBox** instances with code like this:

```
jane := NamedBox new.
jane name: 'Jane'.
jane draw.
```

Figure 13: An unnamed NamedBox named Jane

But the results are not what you might expect (Figure 13). We know that **NamedBox**'s **draw** is going to have to draw the name of the box. **NamedBox** will try to draw Jane as soon as she's created, but her name will still be **nil** (the default value of all new variables). There is code that enables this to work, but the default name is "Unnamed." When the box is then named Jane, it becomes the mishmash of Figure 13. A new way to created **NamedBox** instances should provide a name immediately. A new class method **named:** will create a new instance, and will also set its name from the input argument.

```
named: aName
        | newBox |
        newBox := super new.
        newBox undraw.
        newBox name: aName.
        newBox draw.
        ^newBox
```

For the most part, this is not too surprising of a method: It looks like the same pattern used in **move**, **grow**, **moveTo**, and other methods. The object "undraw's" itself, sets its name, then redraws itself. There are two interesting pieces to note:

- The first line **newBox := super new.** accesses **super**. Super is a predefined special variable that accesses the superclass of the method. In this example, **super** will be **Box**, since **Box** is **NamedBox**'s superclass. This is an explicit call to **Box new**, without using the name **Box**. (Note: **newBox** here is a **NamedBox**, not a **Box**.)
- This doesn't really get around the problem of drawing an unspecified name! **undraw** is going to have to erase the name even if it is not yet defined.

All that's left to define of **NamedBox** now is the new definition of how to **draw** and **undraw NamedBox** instances.

```
draw
        super draw.
        self drawNameColor: (Color black).

undraw
        super undraw.
        self drawNameColor: (Color white).
```

```
drawNameColor: aColor
      | displayName |
      (name isNil) ifTrue: [name := 'Unnamed'].
      displayName := name asDisplayText.
      displayName
            foregroundColor: aColor
            backgroundColor: (Color white).
      displayName displayAt: position.
```

Each of **draw** and **undraw** ask **Box** to do its version of these methods (via the reference to `super`), then draw the name—in black for drawing, in white for undrawing. The interesting part of these methods is **drawNameColor**:.

- The first line of **drawNameColor:** saves the day if the name isn't provided. It explicitly checks if the name exists, and if it doesn't, it's set to **'Unnamed'**. That explains the funny look of Figure 13. But why don't we ever see 'Unnamed' when using the new **named:** method for creating objects? Because in **named:**, the default name is only drawn in white on a white background, via **undraw** before the name is set and **draw** is called.
- We can convert the name from a **String** into a displayable object via **asDisplayText**. We store that new object in the local variable **displayName**.
- **DisplayText** instances can set their foreground and background colors, so we set it up for our white background, with the foreground whatever the argument color is.
- The name is then displayed as **displayName** at the **Box** instances' position.

Exercises: Exploring Classes and Boxes

1. Let's say that **NamedBox**, a subclass of **Box,** also defined a class method **new** as **^super new initialize**. Will this work? Why or why not? What's the downside? (Hint: Print something to the Transcript inside of **initialize**. How many times does it print?)
2. What would happen if the **new** method started with **self new** instead of **super new**?
3. Smalltalk's iteration can be applied to Joe. **10 timesRepeat: [joe move: 10 @ 0]** will move Joe horizontally a total of 100 pixels, in 10 pixel increments. What would

you write to get Joe spinning? To get Joe and Jill spinning at the same time? At different rates of spin?

4. Even though **super** just sends a message up to the superclass, we cannot just replace **super** with the name of the superclass and have everything work okay. **super new** in **NamedBox** is not the same as **Box new**. **super draw** is not the same as **Box draw**. Why not?
5. **Pen**s already know how to display text. Modify **NamedBox** so that it uses the Pen's method for drawing text, rather than using **DisplayText**.

7 Generating: How to Go From "Sample Code" to "Reuse"

There are a variety of neat features in the Box microworld, as well as other examples we're going to be exploring. You will probably want to use them in your own code. But it's not always obvious how to go from a sample piece of code to something that you can make do what you want. Fortunately, Squeak provides lots of tools to help with this process.

Let's say that you want to create an interesting effect on the screen. You want to print your name in a bunch of colors, maybe even random colors. You saw this piece of code in the earlier example, so you know that it should be possible:

```
displayName := name asDisplayText.
displayName
        foregroundColor: aColor
        backgroundColor: (Color white).
displayName displayAt: position.
```

Let's start out by finding the original method that does the color drawing. If you type into a workspace **foregroundColor:backgroundColor:**, you can type Alt-M (Apple-M) to get the *implementors* of this method. There is only one, so you can click on it to see what's going on. Nothing much really. Now, click on the list and choose (yellow button menu) *Senders* (or just type Alt/Apple-N). Now we can see how the code gets used in a variety of situations. There's even an example, a class method in **DisplayText** (Figure 14). You can try it by DoIt on the comment (the line at the bottom in double quotes) **DisplayText example**. It generates an interesting pattern with text. (For this example, note that the comment in the code is actually wrong: You terminate with a mouse click, not a keypress.)

The example shown (**DisplayText example**) as well as the other senders give us a bunch of examples of drawing text with color. We can even see from the implementation of the example that there is a transparent color for a background. However, these examples don't help us generate different colors.

We know that there is a class named **Color**, both from the original sample code and from the example. We can browse the class **Color**. The easy way to get there is to select the word "Color" somewhere and type Alt-B/Apple-B. Choose the class methods of Color, since that's what all the colors we've seen have been. We can see that there are a lot of named colors (click on the method category "named colors"), as well as many ways to generate colors ("instance creation"). One of them, **r:g:b:** just takes three numbers between 0 and 1 to generate a color. That sounds useful for generating random colors.

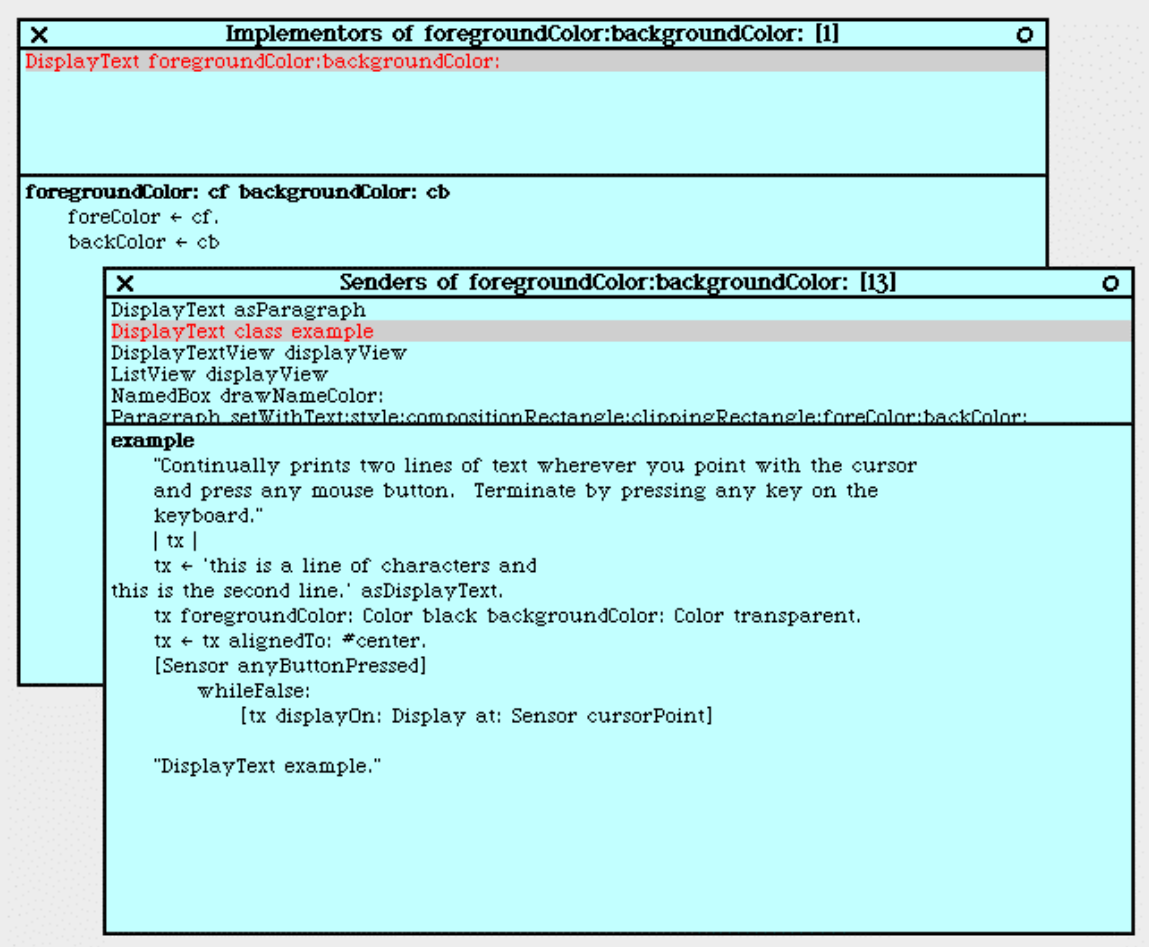

Figure 14: Implementers and Senders of foregroundColor:backgroundColor:

How do we get random things? A good strategy is to look for a class that does what you want. If you choose "Find Class" from the first pane of a Browser, and type Random, you'll find that there is a class named **Random**. That seems to be getting closer, but it isn't clear how to use it.

When facing a new class, the first thing to do is to check the class comment. Click on the "?" button on the browser. In this case, the class

comment on **Random** tells you about how the class is implemented, but not how it's used.

The second thing to try is to look for class methods. There are often examples in class methods that explain how to use classes. In this case, **Random** does have an example method with lots of examples. One of the things it says is:

> The correct way to use class Random is to store one in
>
> an instance or class variable:
>
> myGenerator ← Random new.
>
> Then use it every time you need another number between 0.0 and 1.0
>
> myGenerator next

That sounds useful, since the range is the same as the range expected by **r:g:b:**. Also, there is a way of generating random integers by sending **atRandom** to the highest possible integer. That can be useful to generate random points where to display names.

Here's a piece of workspace code based on all of this:

```
myGenerator ← Random new.
30 timesRepeat: [
    name ← 'Mark Guzdial' asDisplayText.
    name foregroundColor:
            (Color r: myGenerator next
                g: myGenerator next
                b: myGenerator next)
        backgroundColor: (Color transparent).
    name displayAt: (100 atRandom) @ (100 atRandom).]
```

SideNote: You may be wondering why we assign **name** inside the loop. Why don't we create **name** as a **DisplayText** before the loop starts, then just reuse it? It's a loop invariant, isn't it? Try it! You'll find that all the names are the exact same color. **asDisplayText** creates a new **DisplayText** object each time it's called. If we only call it once, we just set the *one* instance to different colors, and at the end, that *one* instance has only *one* final color, stamped lots of times.

This does what we wanted (Figure 15). Through hunting up implementors and senders, and looking through examples and comments in the classes, it becomes pretty easy to build what you want from a good sample piece of code.

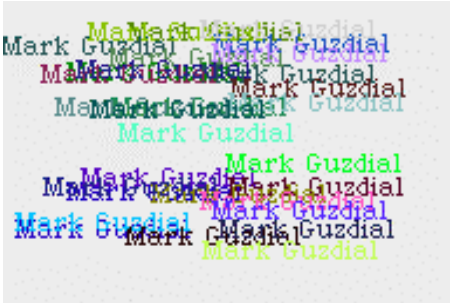

Figure 15: Result of the Random Name Code

8 Improving Boxes: Efficiency, Animation, and Design

The real advantage of an object-oriented structure is the ease with which changes can be made. We can show that feature easiest by actually making some changes to **Box** and **NamedBox**. In this section, we make them more efficient and more amenable to animation. Then, we begin introducing a design perspective on Boxes.

8.1 Drawing Boxes Better

The **Box** and **NamedBox** described above are inefficent and poorly designed. There are two obvious places where the system is flawed:

- **Box**es have a **tilt** and a **position**, and they use **Pen**s for drawing. **Pen**s already have a heading and a location. Why not just create one **Pen** per **Box** instance so that we no longer need the tilt and position variables inside Box?
- **draw** and **undraw** are almost identical pieces of code, which is a bad design. Any changes to how **Box** instances are drawn requires changes to both methods.

Implementing these two fixes requires changing fairly little code. First, we have to redefine the **Box** class so that we have new instance variables. We'll need a **pen** instance variable to hold the **Box**'s **Pen** instance. We'll still need a **size**, but **tilt** and **position** will get passed on to the **Pen**.

```
Object subclass: #Box
        instanceVariableNames: 'size pen '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Boxes'
```

The way that **Box** instances get initialized also has to change in order to use the **Pen**.

initialize

```
pen ← Pen new. "Put a pen in an instance variable."
pen place: 50@50.
size ← 50.
self draw.
```

The various accessor methods also need to change, since some of them now access the Pen instance instead of the Box instance variables. This activity of asking another object to perform a service for the original object receiving the message is called *delegation*. The **Box** instance is delegating some of its services down to the **Pen** instance.

grow: increment

```
self undraw.
size ← size + increment. "This stays the same"
self draw.
```

moveTo: aPoint

```
self undraw.
pen place: aPoint.
self draw.
```

move: pointIncrement

```
self undraw.
pen place: (pen location) + pointIncrement.
self draw.
```

turn: degrees

```
self undraw.
pen turn: degrees.
self draw.
```

Drawing and undrawing also has to change in order to utilize the **new** pen variable. The code becomes much smaller, since a **Pen** instance doesn't have to be created and loaded up with the right values. But while we're at it, it seems like the right time to remove the duplicated code between **draw** and **undraw**. The right thing to do is to create a **drawColor:** method that draws whatever color is needed. This makes **draw** and **undraw** very simple.

draw

```
        self  drawColor: (Color black).

undraw
        self drawColor: (Color white).

drawColor: color
        pen color: color.
        4 timesRepeat: [pen go: size; turn: 90].
```

All of the things that we were asking Joe and Jill to do previously now work just fine. Jane the **NamedBox** won't quite work yet. If we try it, we'll find that the **drawNameColor**: method needs the position of the **Box** instance. We can solve that by delegating to the **Pen**.

```
drawNameColor: aColor
        | displayName |
        (name isNil) ifTrue: [name ← 'Unnamed'].
        displayName ← name asDisplayText.
        displayName foregroundColor: aColor
                backgroundColor: (Color white).
        displayName displayAt: (pen location).
```

Now, all of the previous examples work just fine. From the user-programmer's perspective, nothing has changed in the Box world at all. Yet from the microworld-programmer's perspective, we know that the world has changed considerably.

8.2 Animating Boxes

This next change will impact how the user of the Box microworld sees the system. One of the original uses of the Box microworld was to explore animation. **Box**es were moved and spun on the screen, and one of the activities was to invent a "dance" for the **Box** instances. However, animation requires slightly *slower* performance than modern computers.

Modern computers move so fast that the Squeak boxes can do many operations before the eye can see them. We need to slow them down so that the eye can register their positions before they move again. The delay doesn't have to be much. Motion pictures show 30 frames per second, which means that our eye can register a static image in at least 1/30 of a second. Let's use that number as a reasonable guess.

Creating a delay is really easy. There is a class in Squeak called **Delay** which can create delay objects. We can create them for a certain

amount of time. When the object gets the message **wait** it pauses the processor for the correct amount of time.

Where should we have the Box slow down? The first guess might be in **drawColor:** so that every drawn object would be slowed down. But this turns out to produce really jerky animations. (Please do go ahead and try it.) The reason is that we now use **drawColor**: for erasures, too. We don't need to wait for undraws, just for draws.

If we add a **Delay** creation and wait in draw, the result is very nice.

```
draw

    self  drawColor: (Color black).

    (Delay forSeconds: (1/30)) wait.
```

Try something like this workspace code to see the effect.

```
joe ← Box new
jane ← Box new

30 timesRepeat: [jane turn: 12. joe turn: 10.
                 jane move: 3@4. joe move: 2@3].
```

8.3 Designing Boxes

What we did in Section 8.1 isn't that complicated. We simply moved variables from one object to another. But that may not be obvious to anyone. We could show them the code, but it would be nice if there was a way of describing the objects that we manipulated and how we did it.

One way of describing the two sets of relationships is with *UML (Unified Modeling Language)*. Figure 16 is one way of looking at the original class structure that was in the Box microworld. **NamedBox** is a subclass of **Box**, and **Box** provided **position**, **size**, and **tilt** attributes. We can also see that **NamedBox** inherits a bunch of services from **Box**.

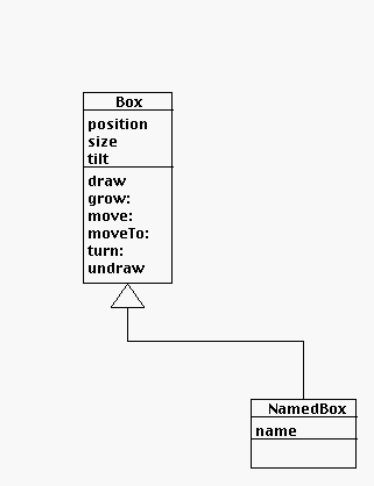

Figure 16: The Original Box Class Structure

Figure 17 is a UML depiction of how the Box microworld was modified. A **Pen** was included as part-of the **Box**. We removed two of the attributes of the **Box** because we were able to delegate them to the **Pen**. We added one attribute to **Box** to keep track of the new **Pen** instance.

The next chapter starts from here: How do we think about designing in objects, and how do we use things like UML to facilitate the designing?

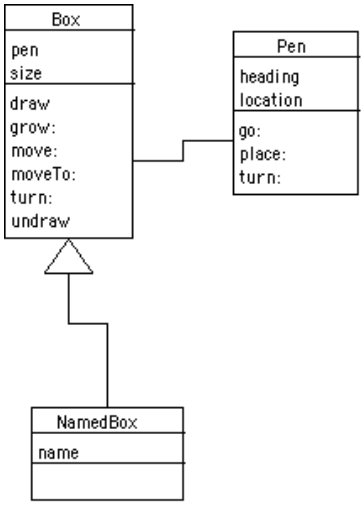

Figure 17: The Modified Box Class Structure

Exercises: More with Boxes and Graphics

6. Try to create a "dance" with the Box microworld. Create two boxes who move around the screen and respond to each other's motions.
7. Write the workspace code to assemble a pyramid of boxes.

References

Joe the Box appeared in Scientific American in:

Kay, A. C. (1977). Microelectronics and the Personal Computer. *Scientific American*(September), 231-244.

Turtle graphics (the way that Pens work) is an amazingly powerful way of looking at mathematics. For a formal and deep treatment, see:

Abelson, H. and A. A. diSessa (1986). *Turtle Geometry: The Computer as a Medium for Exploring Mathematics*. Cambridge, MA, MIT Press.

BitBlt is covered in great detail in the original Smalltalk-80 book:

Goldberg, A. and D. Robson (1983). *Smalltalk-80: The Language*. Reading, MA, Addison-Wesley.