Object Oriented Programming in NeWS

Owen M. Densmore

Sun Microsystems
Mt. View, California

Abstract

The NeWS† window system provides the *primitives* needed to create window managers and user-interface toolkits, but does not, itself, supply either. This is done to achieve a layering strategy for building several higher level systems that can share NeWS as their low level window system. None of the traditional ‘‘tool kit’’ solutions currently span the diverse set of clients NeWS needed to serve; they simply lack sufficient flexibility. We are exploring an object oriented approach which uses a flexible inheritance scheme. This paper presents our initial attempt at introducing a Smalltalk style class mechanism to PostScript‡, and our first use of it.

Introduction to NeWS

NeWS is a server-based window system which replaces the usual network protocols for expressing window and graphics primitives by an interpreted programming language. The language consists of almost all of Adobe System’s PostScript [5], with some extensions.

The extensions to PostScript include:

- Primitives for managing client TCP/IP style connections.
- Primitives for light-weight processes.
- Multiple drawing surfaces called ‘‘canvases’’.
- An event mechanism for handling user input and inter-process communication.
- Use of garbage collection.

See the NeWS Preliminary Technical Overview [4] for details. For the rest of the discussion, use of the word ‘‘PostScript’’ will include these extensions.

The NeWS environment consists of the NeWS window server communicating with client programs using standard TCP/IP. The client does not, however, have to use a fixed protocol for window and graphics primitives. Instead, each client connection has its own ‘‘private’’ light-weight process [LWP in the figure] executing the client’s PostScript commands. This, in effect, replaces a network window protocol with a language.

---

† NeWS is a trademark of Sun Microsystems Inc.

‡ PostScript is a trademark of Adobe Systems Inc.

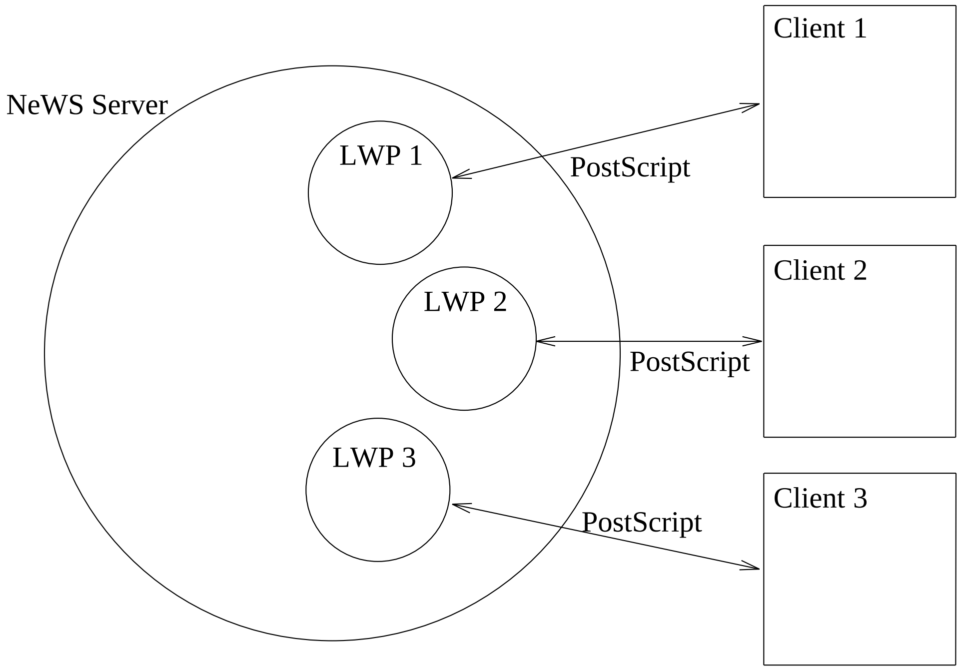

Figure 1 *NeWS Client-Server Environment*

Evolution of NeWS Tools

NeWS provides a rich set of graphics and window manipulation primitives but does not, however, provide window managers or user-interface toolkits. (NeWS is a ‘‘window kernel’’, not a ‘‘window shell’’.) This is done to achieve a layering strategy for building several higher level systems that can share NeWS as their low level window system. A major problem in providing this layer is the extreme diversity of the NeWS user community: OEM’s, the Lisp community, our past SunView clients and X window system clients, to name a few. It became clear that the more traditional user-interface ‘‘tool kit’’ would have difficulty providing sufficient flexibility.

A second problem was *where* to put the higher layer – on the server or in the client. There are several advantages to providing the tools on the server side of the connection:

- Interactive performance would be improved since the user interface code would be kept ‘‘hot’’ in the server.
- Client code size would be decreased by sharing the server process’ code.
- Since the NeWS environment is much more powerful than the typical C client’s, it would allow very rapid prototyping. We show below, for example, that Smalltalk-like classes can be implemented in two pages of PostScript.
- No preference would be given to C clients over others; PostScript would be a common language for all clients. Whenever a new package was made available [by *any* client], it would immediately be available to all clients.

Our first attempt was to define a *package* to be a PostScript dictionary that managed an *object* that was also a dictionary. (Dictionaries are discussed below.) This met with success during our alpha release, especially in terms of flexibility. One Lisp client, for example, built a compiler to translate Lisp window calls into PostScript using the packages delivered with the system. Where they needed to modify the behavior of one of the package procedures, they replaced it with a procedure that called the original one.

This led us to look for a formalization of this style. A Smalltalk-like class mechanism seemed to fill this need. Just before our beta release, therefore, we decided to look for the extensions we would need to make to PostScript to support classes. Much to our surprise, PostScript could implement classes with no modifications! The secret is PostScript dictionaries.

PostScript and Dictionaries

PostScript is a forth-like (prefix notation, stack based) interpretive language developed by Adobe Systems [5]. It is a strongly, but dynamically, typed language. By *dynamically* I mean that object type is determined at run time. It is ‘‘polymorphic’’ in that an object may have different types at different times during execution of a program. In the figure below, for example, *foo* is assigned the number 10 in one context, and the string *abc* in another.

Dictionaries are compound PostScript objects that contain key-value pairs. They can be used in two basic ways. First, their values may be set and retrieved explicitly by the *get* and *put* primitives:

`MyDict /foo 10 put` ..causes the value 10 to be associated with the key foo.
`MyDict /foo get` ..causes the value for the key foo to be put on the operand stack.

Second, their values may be set and retrieved emplicitly by use of the dictionary stack. Simply using a name in PostScript causes that name to be looked up in the set of dictionaries currently on the dictionary stack. The primitive *def* will define a key-value pair in the topmost dictionary, while the primitive *store* will first look to see if the key is defined in the dictionary stack, assigning it to that value if present.

This second form is the basis of PostScript’s name scoping and over-ride mechanism. The set of names (primitives and data) known to the interpreter is the set of names in the current dictionary stack. A name can be re-defined by simply defining it in a higher dictionary.

The figure below shows three clients’ dictionary stacks. All clients have a shared system dictionary and a private user dictionary. Two clients have additional dictionaries on their stacks. (Note: take care not to confuse the dictionary stack with the operand stack which has the operands and results of PostScript operators.) The system dictionary has a name *foo* which was defined at startup. Clients 1 and 3 have not redefined *foo*, thus will share the initial definition of *foo* (as the string ‘‘abc’’). Client 2, on the other hand, has defined *foo* to be the integer 10. It is this capability to over-ride which makes PostScript adaptable to inheritance schemes in general, and to the class mechanism in particular.

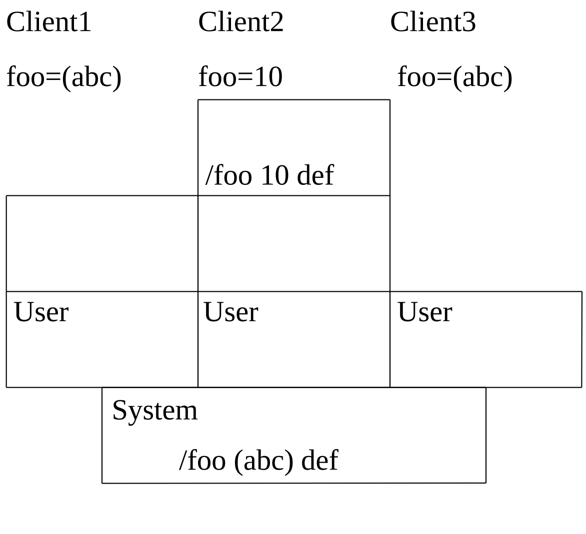

Figure 2 *Multiple Dictionary Stacks*

Modules and Classes

The reader unfamiliar with the use of message-passing, classes, and object-oriented programming may browse through the references listed at the end of the chapter [1, 2, 3]. Many of the essential ideas in class-based systems are similar to the more traditional ‘‘package’’ or ‘‘module’’ based systems, however.

Briefly:

- Packages (modules) are replaced by *classes*.
- Procedures in packages are replaced by *methods* in classes.
- Creating package objects is replaced by creating new *instances* of a class.
- Package local and global variables are replaced by *class variables*.
- Object variables are replaced by *instance variables*.

New notions are:

- Classes are ordered into a hierarchy by *subclassing* a new class to a prior one, *inheriting* its methods, instance variables, and class variables.
- Methods are invoked by use of the *send* primitive. The term *message* is used for an invocation of a method with its arguments.
- There is a means of constructing classes that is absent in most languages’ module creation.
- Two new concepts, the *self* and *super* pseudo-variables are introduced. They are used in methods to refer to the object sent the method, and the method’s superclass, respectively. Note: self does *not* refer to the method’s class, but rather to the object that originally caused the method to be invoked.
- Unlike PostScript procedures, methods are *compiled* when a class is created. Currently this simply resolves self and super, and performs some minor optimizations.

The relationship between an instance and its class and superclass is shown in the figure below. We have made an instance, *aFoo*, of class *Foo* which is a subclass of class *Object*. An instance has a copy of all instance variables of its superclasses, thus *aFoo* has those required by both *Foo* and *Object*. The methods known by an instance are stored in the classes in its superclass chain. Thus *aFoo* can only respond to methods residing in *Foo* and *Object*.

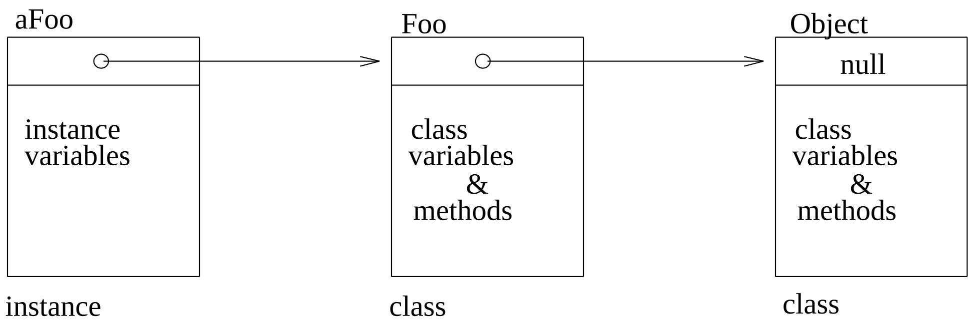

Figure 3 *Relationship between Instances and Classes*

Sending a message to an instance requires packaging the arguments to the method, finding the method in the class chain, invoking the method in the proper context, and possibly returning a result to the sender. If the pseudo-variable *self* is used for the object in sending a message, the search for the method starts at the beginning of the chain, while if *super* is used the search starts in the superclass.

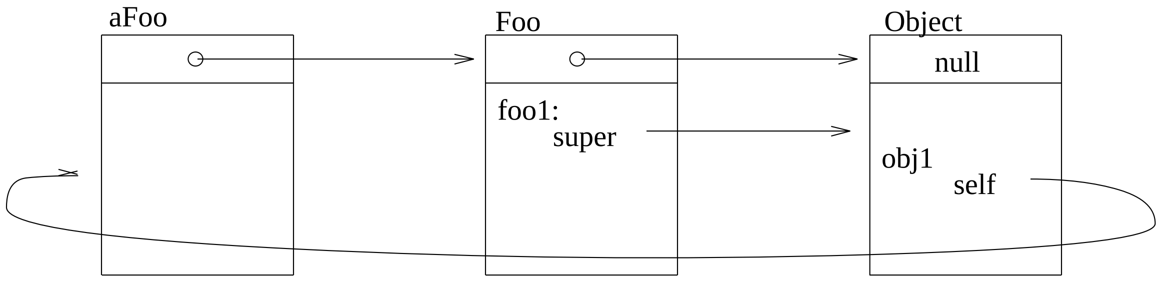

Figure 4 *Self and Super*

PostScript Classes

The PostScript implementation of classes uses dictionaries to represent the classes and instances. Instances contain all the instance variables of all their superclasses. Classes contain their methods as PostScript procedures. Our current implementation of class is entirely in PostScript and is given in Appendix A.

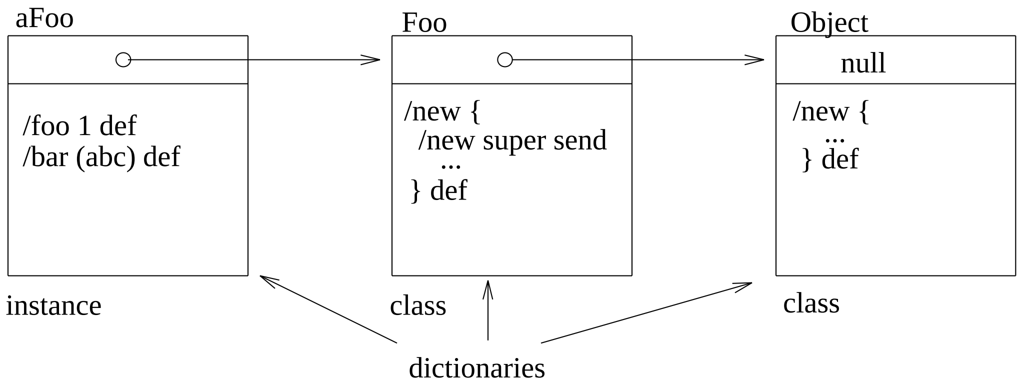

Figure 5 *PostScript use of Dictionaries as Objects*

Classes are built using the *classbegin* . . . *classend* procedures; messages are sent with the *send* primitive:

- **classbegin**: classname superclass instancevariables => –
  Creates an empty class dictionary which is a subclass of *superclass*, and has *instancevariables* associated with each instance of this class. The dictionary is put on the dict stack. *Instancevariables* may be either an array of keywords, in which case they are initialized to *null*, or a dict, in which case they are initialized to the values in the dict.

- **classend**: – => classname dict
  Pops the current dict off the dict stack (put on by *classbegin* and presumably filled in by subsequent *def*s), and turns it into a true class dictionary. This involves compiling the methods and building various data structures common to all classes.

- **send**: <optional args> method object => <optional results>
  Establishes the object’s context by putting it and its class hierarchy on the dictionary stack, then executes the method. The method is typically the keyword of a method in the class of the object, but it can be a arbitrary procedure (see the examples below).

The *send* primitive establishes the context for execution of the method by adding the instance and its superclass chain to the dictionary stack. It then executes the method within this context. The arguments to the method will be on the operand stack as in typical PostScript procedure calls.

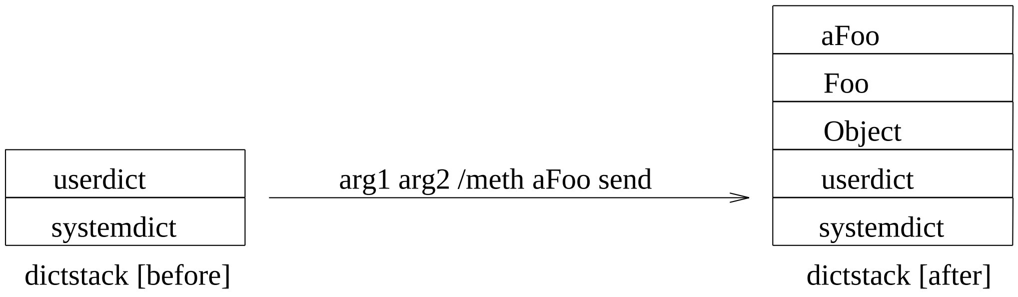

Figure 6 *PostScript use of Dictionaries as Objects*

Example: Class Foo

Here we build a sample class, *Foo*:

```
/Foo Object
dictbegin                   % (initialized) instance variables
    /value 0 def
    /time null def
dictend
classbegin
    /Value [0 1] def        % class variables
    /Time currenttime def

    /new {                  % class methods
        /new super send begin
            /resettime self send
            currentdict
        end
    } def
    /printvars {
        (..we got: Value=%, value=%, Time=%, time=%.\n)
            [Value value Time time] printf
    } def
    /changevalue { % value => - (changes the value of "value")
        /value exch def
    } def
    /resettime { % - => - (sets time to the current time)
        /time currenttime def
    } def
classend def
```

Foo is a subclass of *Object*, discussed above. *Foo* has two instance variables unique to each of its objects; *value*, an arbitrary value associated with the object, and *time*, the time of creation of the object. They are initialized by use of the dict form of specifying instance variables. (The *dictbegin* . . . *dictend* pair are standard utilities which create a dict just the right size for its *def*s.) Similarly, *Foo* has two class variables; *Value*, an arbitrary value associated with the class, and *Time*, the time of creation of the class. Note that *Time* is initialized by calling the PostScript primitive *currenttime*.

Foo has four methods: *new*, *printvars*, *changevalue* and *resettime*. *new* first calls its super class to get a raw instance, which it then initializes by setting the time to the current time. Note the use of *begin* . . . *currentdict end*. This is a common cliché. Also note the use of both self and super: we ask our superclass to make a new instance of itself and initialize it, then ask self to reset our time. *printvars* is used to print the instance and class variables of the object; note how this uses another standard utility, *printf*. *changevalue* is a method which takes a single argument and assigns it to the instance variable *value*. Finally, *resettime* sets the instance variable *time* to the current time.

Using Class Foo

Let's look at some uses of *Foo*: Here we create a new instance, *foo* of *Foo*. We then print out its initial values, shown by the line starting with ''`..we got`''.

```
/foo /new Foo send def
/printvars foo send
..we got: Value=array[2], Value=0, Time=1.728, time=4.042.
```

Now we are going to change the value of *foo*'s instance variable *value*. Note that *value* initially was an integer, and we are changing it to a string – an example of PostScript ''polymorphism''.

```
(Value) /changevalue foo send
/printvars foo send
..we got: Value=array[2], value=0, Time=11.95, time=12.25.
```

Now we do an odd thing, we simply send an executable array (a procedure) to *foo*. The effect of doing this is to execute the procedure within the context of *foo*. (This is somewhat unfair, like cutting paper in Origami, but nicely illustrates the flavor of our combination of PostScript language features and our class extensions.) The procedure we're sending to *foo* is *{Value changevalue}* which assigns *Value*, the class variable, to *value*, the instance variable.

```
{Value changevalue} foo send
/printvars foo send
..we got: Value=array[2], value=array[2], Time=11.95, time=12.25.
```

The above example did not go through method compilation, thus ''self'' and ''super'' could not be used. Let's send an executable procedure to *foo* to change *value* to the current value of *Time*, but this time using the more orthodox *doit* method which *does* go through method compilation. The argument to *doit* first sends the message *Time* to itself, which simply returns the value of the *Time* class variable. This then gets sent to *changevalue*. The result is:

```
{/Time self send /changevalue self send} /doit foo send
/printvars foo send
..we got: Value=array[2], value=11.95, Time=11.95, time=12.25.
```

Next we ask *foo* to reset its time value by using the method *resettime*:

```
/resettime foo send
/printvars foo send
..we got: Value=array[2], value=11.95, Time=11.95, time=13.16.
```

Now let's change the class variable *Value* of class *Foo* by use of the *set* method. Because we are not using any of the context of the class, we can send a simple array:

```
[/Value null] /set Foo send
/printvars foo send
..we got: Value=null, value=11.95, Time=11.95, time=13.16.
```

Here we set *Value*, but using the class variable *Time*. This requires deferring the evaluation until within the context of the class, thus use of an executable array:

```
{/Value Time} /set Foo send
/printvars foo send
..we got: Value=11.95, value=11.95, Time=11.95, time=13.16.
```

As a final example, see Appendix B for the PostScript version of the self and super tests on page 62 to 65 in [3]. We used this to convince ourselves that we were implementing self and super correctly!

Class Items

The first real use of classes in NeWS was the class Item. Items are simple, graphical input controls, such as buttons and check-boxes. The Item hierarchy consists of the base class Item, the subclass LabeledItem, and several practical subclasses of LabeledItem. Both Item and LabeledItem provide no usable instances themselves. Rather they are abstract superclasses used to provide a common link for subclasses.

Class Item has these major components:

- A canvas used to depict the item and to be the target of the item’s input.
- A set of procedures for painting the canvas and handling activation and tracking events.
- A current value and a procedure which notifies the client when that value changes due to action of the tracking procedures.
- Methods for creating, moving and painting the item, and for returning the item’s location and bounding box.

The definition of the class Item is given in Appendix C. Rather than discuss the class in detail, we look at two sample implementations: SampleToggle and SampleSlider, and a simple program that uses them both.

SampleToggle and SampleSlider

SampleToggle provides tracking by implementing the client’s Down, Up, Enter, and Exit procedures. The ItemValue is treated as a boolean, with *true* meaning *on*. Down and Enter simply assign `not ItemInitialValue` to ItemValue, while Exit resets it to ItemInitialValue. Up simply calls the notify procedure if the state has changed. SampleToggle adds no instance or class variables.

Here are two toggles, one on and one off, and the implementation of the class SampleToggle:

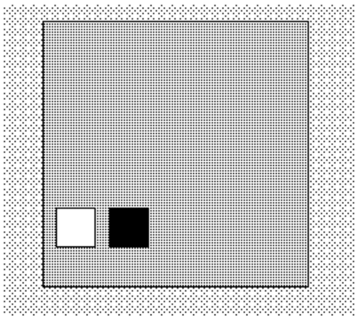

Figure 7 *Two Instances of Class SampleToggle*

```
/SampleToggle Item []
classbegin
    /new { % initialvalue notifyproc parent width height => item
        /new super send begin
            /NotifyUser exch cvx def
            /ItemValue exch def
            currentdict
        end
    } def

    /PaintItem {
        ItemValue
            {0 fillcanvas}
            {1 fillcanvas 0 strokecanvas} ifelse
    } def
    /ClientDown {ItemInitialValue not SetToggleValue} def
    /ClientUp {ItemValue ItemInitialValue ne {NotifyUser} if} def
    /ClientEnter {ClientDown} def
    /ClientExit {ItemInitialValue SetToggleValue} def

    /SetToggleValue { % value => - (set value & paint toggle)
        /ItemValue exch store
        /paint self send
    } def
classend def
```

SampleSlider provides tracking by implementing the client's Down, Up, and Drag procedures. The Down and Drag procedure are identical, simply projecting the current x coordinate of the mouse onto the slider.

Here is a slider and its implementation:

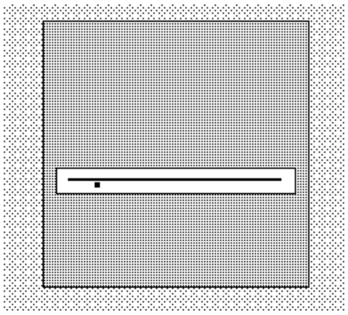

Figure 8 *An Instance of Class SampleSlider*

```
/SampleSlider Item [/SliderX /SliderY /SliderWidth /SliderHeight]
classbegin
    /new { % initialvalue notifyproc parent width height => item
        /new super send begin
            /NotifyUser exch cvx def /ItemValue exch def
            /SliderX            ItemHeight 2 div 1 sub def
            /SliderY            ItemHeight 2 div def
            /SliderWidth        ItemWidth ItemHeight sub def
            /SliderHeight       2 def
            currentdict
        end
    } def
    /PaintItem {
        ItemCanvas setcanvas 1 fillcanvas 0 strokecanvas
        SliderX SliderY SliderWidth SliderHeight rectpath fill
        ItemValue 0 PaintSliderValue
    } def
    /ClientDown {
        SetSliderValue
        ItemValue ItemPaintedValue ne {
            ItemPaintedValue 1 PaintSliderValue
            ItemValue        0 PaintSliderValue
        } if
    } def
    /ClientUp {ItemValue ItemInitialValue ne {NotifyUser} if} def
    /ClientDrag {ClientDown} def
    /PaintSliderValue {  % value gray => -
        setgray
        SliderX add SliderY 5 sub 4 4 rectpath fill
        /ItemPaintedValue ItemValue store
    } def
    /SetSliderValue {
        /ItemValue
            CurrentEvent geteventlocation pop SliderX sub
            0 max SliderWidth min store
    } def
classend def
```

Below is a test program that uses these samples. The notify procedure prints the value of the item using the printf utility. We start by building a canvas and painting it. Then we make two items, a button and a slider, putting them in a dictionary called *items*. We then paint them and fork an activation event manager.

Here's what the test looks like, and its implementation:

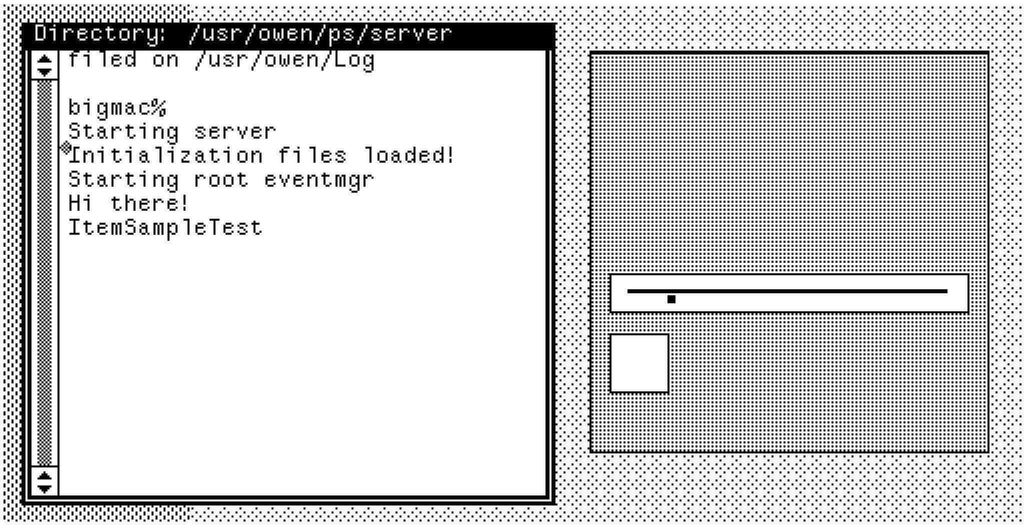

Figure 9 *The Sample Test Program*

```
/ItemSampleTest {
    /notify {ItemValue (ItemValue: % \n) printf} def
    /itembackground .75 def
    /can framebuffer 200 200 createcanvas def
    can setcanvas 200 100 movecanvas currentcanvas mapcanvas
    itembackground fillcanvas 0 strokecanvas

    /items 10 dict dup begin
        /sampletoggle
            false /notify can 30 30 /new SampleToggle send def
            10 30 /move sampletoggle send
        /sampleslider
            20 /notify can 180 20 /new SampleSlider send def
            10 70 /move sampleslider send
    end def
    items paintitems /p items forkitems def
} def
```

After pushing the toggle and sliding the slider, we have:

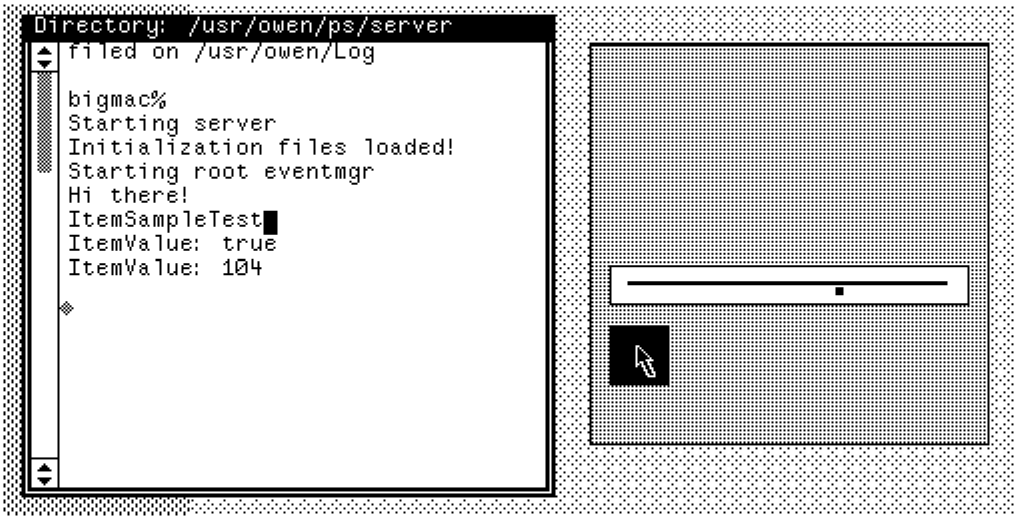

Figure 10 *Use of The Sample Test Program*

What to notice here is the simplicity of the code, emphisizing the power, both of the program and sample items, and of the NeWS programming environment. The implementation of the items and test program in the interpretive NeWS environment took *very* little time.

Class LabeledItem

Most items are more elaborate than the preceeding examples. Class LabeledItem implements a more common item; one that has:

- A polymorphic label-object pair, either of which may be a string, an icon, or a general PostScript procedure.
- A ‘‘round rectangle’’ frame enclosing the item.
- Simple layout rules for automatic positioning of the label and object. The object positon may be to the Right, Left, Top, or Bottom of the label.

The current subclasses of class LabledItem are:

- **ButtonItem**: provides a simple activation/confirmation item
- **CycleItem**: provides check boxes and choices
- **SliderItem**: provides a continuous range of values
- **TextItem**: provides a type-in area
- **MessageItem**: provides an output area
- **ArrayItem**: provides an array of choices

The (abbreviated) class definition is given in Appendix D.

This window contains one of each of these items:

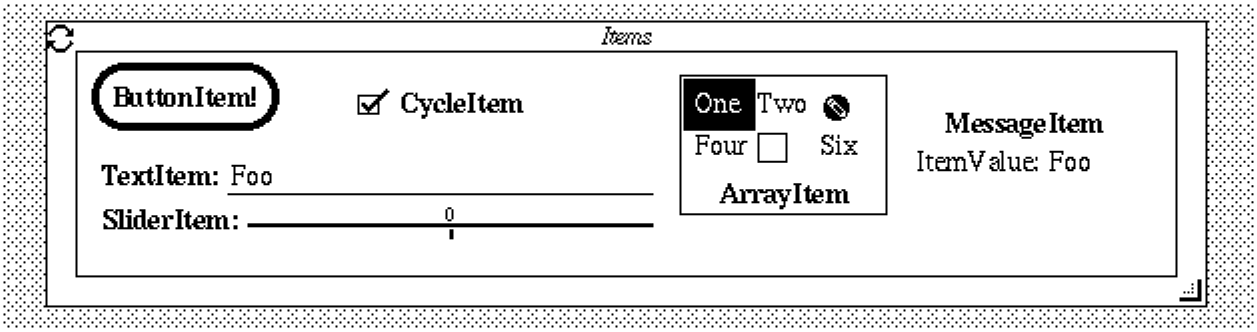

Figure 11 *Subclasses of LabeledItem*

All four object positions are visible in the figure: The text and slider items use */Right*, the cycle item uses */Left*, the message item uses */Top*, and the array item uses */Bottom*. The complete class definition for one of these, CycleItem, is given in Appendix E.

Summary

Use of a Smalltalk-like class mechanism proves to have several advantages for NeWS:

- Classes are a well documented standard discussed in several easily obtainable books.
- Classes formalize the flexibility needed by the NeWS community.
- There are at least two well-documented class hierarchies for application writing: Smalltalk itself, and MacApp, Apple's ''extensible application.''
- Classes are easily and naturally implemented in PostScript.
- Classes offer rapid prototyping and high productivity.

Our initial implementation of classes was done in PostScript itself, with no extensions required to the interpreter. We then implemented a reasonably complex class hierarchy for standard user interaction items. PostScript's polymorphism proved quite useful for yielding different results for different items. For example, TextItems return PostScript strings, SliderItems return an integer, and ArrayItems return a 2 element array index. We feel our initial use of classes has been successful. We plan to use classes for the package level of NeWS.

References

1. Kurt J. Schmucker, *Object-Oriented Programming for the Macintosh*, Hayden Book Company, 1986.

2. *Byte Magazine*, special issue on *Object-Oriented Languages*, August, 1986, McGraw-Hill Inc.

3. Adele Goldberg and David Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, May, 1983.

4. *NeWS Preliminary Technical Overview*, Sun Microsystems, October 2, 1986.

5. Adobe Systems Incorporated, *PostScript Language Reference Manual*, Addison Wesley, July, 1985.

Appendix A: Complete listing of PostScript class implementation

```
%
%  Objects 'n stuff.
%

/ObjectTemplate dictbegin  % All objects have these fields:
    /parentDict      null def  % link to my parent dict; stops at null.
    /parentDictArray null def  % complete parent chain to Object!
dictend def
/ClassTemplate dictbegin  % Class objects have these fields in addition:
    /instanceVars     null def % this class' additional inst vars
    /instanceVarDict  null def % this class' total inst vars
    /instanceVarExtra 10 def   % extra space for class var over-rides
    /className        null def % name of the class (as a keyword)
    /subClasses       nullarray def  % subclass list (for browsing)
dictend def

% Create a sub-class of the given class.
%    The instancevariables may be either an array or a dict.
%    The advantage of using a dict is that the variables will be
%    pre-initialized to a value you chose, rather than "null".
/classbegin {   % classname superclass insvars => - (newclass on dict stack)
dictbegin
    ObjectTemplate {def} forall
    ClassTemplate {def} forall

    /instanceVars exch def
    /parentDict exch def
    /className exch def
} def
/classend { %  - => classname newclass
    currentdict {
        dup xcheck {parentDict methodcompile def} {pop pop} ifelse
    } forall

    /instanceVarDict instanceVars def
    /parentDictArray [] def

    instanceVarDict type /arraytype eq {
        /instanceVarDict instanceVarDict length dict dup begin
            instanceVarDict {null def} forall
        end def
    } if

    parentDict null ne {
        parentDict /subClasses 2 copy get [className] append put
        /instanceVarDict
            parentDict /instanceVarDict get instanceVarDict append def
        /parentDictArray
            parentDict /parentDictArray get [parentDict] append def
    } if

    className
dictend
} def
% Crack open the methods and fix for "super send" and "self send"
/methodcompile { % method parentdict => newmethod
10 dict begin
    /superpending false def
    /selfpending false def
```

```
    /parentDict exch def
    [ exch {
        dup /send eq superpending selfpending or and {
            pop pop
            superpending
                {parentDict /className get cvx /supersend cvx}
                {cvx} ifelse
        } if
        dup type /arraytype eq {parentDict methodcompile} if

        dup /super eq /superpending exch def
        dup /self eq /selfpending exch def
    } forall
    ] cvx
end
} def

% Generic Smalltalk-ish Primitives.
% Send a message to an object.
/send { % <args> message object => <results>
    dup /parentDictArray get {begin} forall
    begin
        cvx exec
    parentDictArray length 1 add {end} repeat
} def
% Send a message to super without poping myself.
/supersend { % <args> keywordmessage superclass => <results>
    exch { 2 copy known {exit} {exch /parentDict get exch} ifelse } loop
    get exec
} def
% Put me on the operand stack.
/self {/parentDict where pop} def

% Your basic object!
/Object null [] classbegin
    /new { % class  =>  instance
        ObjectTemplate length instanceVarDict length instanceVarExtra
        add add dict dup begin
            instanceVarDict {def} forall
            ObjectTemplate {def} forall
        end
        dup /parentDict currentdict put
        dup /parentDictArray parentDictArray [currentdict] append put
    } def
    /doit { % proc ins  =>  - (compile & execute the proc)
        parentDict /parentDict get methodcompile exec
    } def

    /set { % {/key value ...}  =>  -  stores the values in the object
        mark exch cvx exec
        counttomark 2 div {def} repeat pop % store??
    } def
classend def
```

Appendix B: Smalltalk [3] page 62 example in PostScript

This is a PostScript implementation of the self and super tests given in [3] on pages 62 - 65.

```
/smalltalkpage62 {
    /One Object [] classbegin
        /test {1} def
        /result1 {/test self send} def
    classend def

    /Two One [] classbegin
        /test {2} def
    classend def

    /ex1 /new One send def
    /ex2 /new Two send def

    /test ex1 send =
    /result1 ex1 send =
    /test ex2 send =
    /result1 ex2 send =

    /Three Two [] classbegin
        /result2 {/result1 self send} def
        /result3 {/test super send} def
    classend def
    /Four Three [] classbegin
        /test {4} def
    classend def

    /ex3 /new Three send def
    /ex4 /new Four send def

    /test ex3 send =
    /result1 ex4 send =
    /result2 ex3 send =
    /result2 ex4 send =
    /result3 ex3 send =
    /result3 ex4 send =
} def
```

Results:

```
smalltalkpage62
1 1 2 2 2 4 2 4 2 2
```

Appendix C: Class Item

```
/Item Object [
% instance variables
    /ItemWidth       % item’s width,
    /ItemHeight      % ..& height,
    /ItemParent      % ..& parent canvas (from "new")
    /ItemCanvas      % the canvas we created for the item
    /ItemValue       % the canvas’ current value
    /ItemInitialValue % the value it started out with
    /ItemPaintedValue % the value it currently shows
    /StartInterest   % the interest which activates the item
    /ItemInterests   % interests used to track item
    /ItemEventMgr    % ..the tracking process
    /NotifyUser      % the user’s notify proc
] classbegin
% default variables
    /ItemFont          DefaultFont def    % the item’s font
    /ItemTextColor     0 0 0 rgbcolor def % ..& text color
    /ItemBorderColor   ItemTextColor def  % ..& border color
    /ItemFillColor     1 1 1 rgbcolor def % ..& background color
% class variables; mainly the std client procs
    /PaintItem   nullproc def  % the core of the /paint method
    /ClientDown  nullproc def  % procedures installed in
    /ClientDrag  nullproc def  %   the activated (tracking)
    /ClientEnter nullproc def  %   process
    /ClientExit  nullproc def
    /ClientKeys  nullproc def
    /ClientUp    nullproc def
    /StopOnUp?   true def      % deactivate on up event?
% methods
    /new              % parentcanvas width height => instance
    /makecanvas       % - => -
    /makeinterests    % - => -
    /move             % x y => -  (Moves item to x y)
    /moveinteractive  % items backgroundcolor => -
                      % (interactively moves the item)
    /paint            % - => -  ([Re]paints item)
    /location         % - => x y
    /bbox             % - => x y w h
classend def
```

Appendix D: Class LabeledItem

```
/LabeledItem Item
dictbegin
% instance variables
    /ItemObject     nullstring def % The item’s "object"
    /ObjectX        0 def          % and bounding rect:
    /ObjectY        0 def
    /ObjectWidth    0 def
    /ObjectHeight   0 def
    /ItemLabel      nullstring def % The item’s "label"
    /LabelX         0 def          % and bounding rect:
    /LabelY         0 def
    /LabelWidth     0 def
    /LabelHeight    0 def
    /ItemBorder     2 def    % Extra space around the item
    /ObjectLoc      null def % Label-Object position
    /ItemGap        5 def    % Distance between object & label
    /ItemFrame      0 def    % Draw frame if not zero
    /ItemRadius     0 def    % Radius of frame
dictend
classbegin
% default variables
    /ItemLabelFont  Item /ItemFont get def
% class variable: over-ride of PaintItem
    /PaintItem       % - => -
% methods: over-ride new
    /new  % label obj loc notify parent width height => instance
% utilities used to manipulate label-object pair
    /LabelSize       % - => width height
    /ShowLabel       % - => -
    /ShowObject      % - => -
    /EraseObject     % - => -
    /AdjustItemSize  % - => -
    /CalcObj&LabelXY % - => -
classend def
```

Appendix E: Class CycleItem

```
/CycleItem LabeledItem
dictbegin
    /ItemValue       0 def
    /EraseToUpdate   true def % erase when switching state
    /Cycle           nullarray def
dictend
classbegin
    /new { % label array loc notify parent width height => instance
        /new super send begin
            /Cycle ItemObject def
            currentdict
        end
    } def
    /makecanvas {
        BindCycleObject
    % calculate Label & Object Height & Width:
        Cycle { % calculate bbox for all the cycle objects
            ItemFont ThingSize
            /ObjectHeight exch ObjectHeight max def
            /ObjectWidth exch ObjectWidth max def
        } forall
        LabelSize /LabelHeight exch def /LabelWidth exch def

        AdjustItemSize
        CalcObj&LabelXY

        /ItemCanvas ItemParent ItemWidth ItemHeight createcanvas def
    } def
    /PaintItem {/PaintItem super send false PaintCycle} def
    /SetCycleValue { % Bump? => - (Set ItemValue to initial or bumped value)
        /ItemValue ItemInitialValue 3 -1 roll
            {1 add Cycle length mod} if store
        ItemValue ItemPaintedValue ne {
            true PaintCycle
            /ItemPaintedValue ItemValue store
        } if
    } def
    /BindCycleObject {/ItemObject Cycle ItemValue get store} def
    /ClientDown {true SetCycleValue} def
    /ClientUp {ItemValue ItemInitialValue ne {NotifyUser} if StopItem} def
    /ClientEnter {true SetCycleValue} def
    /ClientExit {false SetCycleValue} def

    /PaintCycle { % updating? => -
        EraseToUpdate and {EraseObject} if
        BindCycleObject ShowObject
    } def
classend def
```