Implementation of an Object-Oriented Shell

James B. Rudolf
Timothy A. Budd
Computer Science Department
Oregon State University
Corvallis, OR 97331-3902

89-60-19

Implementation of an Object-Oriented Shell

James B. Rudolf
Timothy A. Budd

Computer Science Department
Oregon State University

August 30, 1989

1 Introduction

As computers continue to infiltrate our society and become a part of everyday life, the makeup of the user community is changing. Unlike the old days when computers were used exclusively by scientists and computer programmers, we can no longer expect users to be computer literate or to have a certain level of education. And as processing speeds continue to increase, attention is being drawn away from raw MIPS capability and instead aimed towards the necessary dialog between user and machine [Fis89]. As software complexity increases, the computer becomes not only a computational engine but also a communication vehicle. These events make it necessary to redefine how an interface should interact with this changing user group.

Since concern over the human-computer dialog has increased fairly recently, much of the existing software was written assuming a certain amount of expertise. The Unix operating system is an excellent example. It was developed in a research environment by programmers who wanted a powerful software development facility. To the uninitiated, Unix commands can be terse and not always intuitive. Limited and equally terse online documentation is not enough to help a naive user. As a result, this difficulty of use has been one of the biggest complaints about Unix: it is productive only for those that have used it enough to know their way around.

This perceived "unfriendliness" is unfortunate, because Unix is becoming the de facto standard for workstations and even some mainframes in the engineering community, after having been created on a minicomputer and spending much of its evolution in academia. This popularity is also spilling over into the business and non-technical sectors. Many people with little or no computer experience are now working on Unix systems, and finding them difficult to use.

There are two apparent solutions to this problem of user-hostility: discard Unix for a more friendly alternative, or find a way to make Unix more palatable to the less computer literate masses. The first solution simply will not occur. No clearly superior operating system exists that can fill the same niche, and even if one did, the popularity of Unix is growing steadily and it is a powerful environment.

Without changing the operating system itself and creating yet another version of Unix, we can construct an interface that will sit between Unix and the end user. What is needed is an interface that will translate a terse Unix dialog into a dialog more meaningful to casual users. This might include hiding the portions of Unix that complicate its use, and letting the user enter understandable commands that the interface will map onto the corresponding Unix commands.

We have looked into creating such an interface. We hope that by combining graphics with object-oriented concepts, we can create an interface that takes some of the complexity out of working with Unix, without sacrificing the features that make it powerful. We will attempt to bridge the gap between Unix and the end user with a graphical object shell.

In this paper we review the work that went into the interface definition for the object shell, also called OSH. After taking a brief look at some existing interfaces, we will discuss the look and behavior of the object shell. On some occasions we will describe how a feature would look to the user even though the feature was not implemented. We then take a look at the implementation, because the flexibility of the object shell is due in large part to how it was designed. Finally, we list directions that further research could take, and comment on what we have accomplished and learned up to this point.

2 Background

2.1 Existing Interfaces

In the last few years, much attention has been paid to the interface separating the user and the computer. Graphical interfaces are becoming more common, so we mention a handful of graphical interfaces here for comparison purposes.

Suntools and Cedar

Suntools is a popular kernel-based windowing system available from Sun Microsystems [Sun86]. It creates a graphical interface on top of Unix and provides graphical equivalents to certain Unix tools, such as *mail* and *dbx*. It also provides a *shelltool* which is a shell running within a window. Thus, multiple windows can be running multiple shells at once so a user can be working on multiple tasks at once.

A Suntools icon represents a "closed" window or application. It is closed in the sense that the task represented by a window is now an icon and thus using less screen real estate, though the process represented by the window continues to execute while it is in iconic form. The icon has no other meaning in the Suntools definition.

The Cedar programming environment [Tei84], developed at Xerox PARC, is not a front end for Unix as Suntools is, but it uses icons in a similar fashion. When windows are closed, they are stored as icons along the bottom of the screen. As with Suntools, the icon merely saves screen space; the process that it represents continues to execute.

We suggest that while these interfaces are graphical, they (and others like them) are not much of a conceptual departure from the traditional command line interface. They use windows to allow multiple ASCII terminal sessions and an easier-to-use front end for some existing text-based tools, but they don't really have any object-oriented qualities.

The Macintosh Finder

The Apple Macintosh introduced the first commercially successful graphical interface along with the first taste of object-oriented behavior in an interface. Using techniques developed in large part at Xerox PARC, Apple integrated windows, menus, icons, and mice to offer an alternative to the command line interpreter.

In addition to an icon representing a closed window, the Macintosh Finder uses icons to represent files, and introduced the concept of files as *objects*. These objects are divided into a few discrete classes, and only certain operations can be performed on an object of a given class. Some commands even display polymorphism (unbeknownst to most users) by offering similar menu items (such as *print*) for different types of objects. This was the first glimpse of an object-oriented model in a commercial user interface.

The WISh Interface

Some current work being done by Beaudouin-Lafon has experimented with an object-oriented interface for Unix [Bea89]. In his Window Icon Shell interface (WISh), objects not only represent Unix files, but they can also represent users, processes, or remote machines. Beaudouin also introduced the *electric icon*, which is an icon that initiates a process if another icon is dragged over it. For example, to print a textual object, Beaudouin would drag that object over the printer object. While this technique offers objects in a number of classes, it looks more like passing an object as an argument to a process instead of passing a message (representing a process) to an object.

Open Look

The Open Look interface definition is a joint project between Sun and AT&T [SuO89], that is not an actual piece of software but a definition of how an interface should look and behave. Some of its behavior is similar to that of the WISh interface described above.

Open Look attempts to create a completely integrated desktop, so that objects from one application can be passed to another application, which will know what to do with the object based on its type. If you are editing a document and you wish to edit a new one, you can simply drag the icon representing the new document from the file manager into the editor. Or if you receive a mail message with a meeting announcement (in a standard format), drag the message object over your calendar tool, and your calendar will be updated with the time and date of the meeting.

2.2 A Different Approach

This paper and the work supporting it are a further investigation of work done by Tim Budd [Bud89], in which he discussed the rationale for developing a graphical interface for Unix that also had an object-oriented feel throughout. Though he gave a textual object-oriented shell as an example, he considered the benefits of a graphical representation for his shell. The following is a brief review of the more important points from his paper that helped us to define our interface.

A Graphical Interface

We have seen how the interface of the Macintosh has made computers accessible to a wider audience. We also want to use a graphical representation to hide as much system complexity as possible from the user.

A logical approach, based on some existing systems, is to use icons to represent files. A window can represent a Unix directory, and the icons in the window represent the files in that directory. Different buttons on a mouse can be used to perform different functions. One button can be used to display a menu. The items within this menu will vary depending upon the type of object the cursor is over. These menu items will be the set of valid commands that can be performed on the particular type of object. The user will not have to remember the exact name of the command or where that command resides.

In situations when specifying a command is not enough, the graphical interface will also supply dialogs in order to transfer information (such as the options common to many Unix commands) between user and machine. Another mouse button can be used to select icons to be passed as the arguments of a command.

Object-Oriented Design

As object-oriented programming becomes better understood, we are learning how it can help us design and develop applications faster than before because of reuseable code and the fact that some real-world situations can be more accurately modeled using an object-oriented approach [Bud87].

There is no reason why the same properties cannot lead to a cleaner command interpreter. The characteristics of object-oriented design can help to make an interface that is easier for naive users to comprehend and that reuses existing pieces of the system when possible. Specifically, these features of object-oriented design can benefit our interface:

- By defining all Unix files as objects, we can divide them into classes and therefore define a structure and behavior for each file. All objects of the same class will share the same structure (though they may have different values) and behavior. The behavior of our Unix file is defined by the set of valid operations that can be performed on the file and all other files of the same type.
- Data encapsulation protects an object from a dangerous world. The data (or *instance variables*) of an object cannot be directly accessed or modified by other objects. Only the valid operations (or *methods*) defined for the object's class can access the instance variables. Therefore, some level of protection is provided, since the methods control what can be done to an object's data. Another object cannot print a text file. It can only ask the text file to print itself.
- When a new class is defined, it is done in terms of an existing class. An object in the new class inherits the structure and behavior of the existing class, and possibly build on that structure and behavior. This concept of *inheritance* makes reuseable code possible, so that every subclass of a text class can use the *print* method inherited from the text class, or create a more specific *print* method that will override the method in the text class.
- Message passing provides an insulating layer between what the user requests and what needs to be done to satisfy that request. Polymorphism lets the user concentrate on what she wants to accomplish, without worrying about how the task is actually carried out. By specifying that she wants to print a text file, all she needs to remember is *print*. The particular method that is invoked will take care of the details of printing.

As we saw above, the Macintosh has a somewhat object-oriented flavor. The proposed interface could go even further than the Mac by supporting inheritance, message passing, and by allowing the creation of abstract data types to represent Unix files.

A Difference in Philosophy

Hopefully, this interface will hide some of the complexities of Unix from the user. But the philosophies of Unix and object-oriented design are different enough that creating a successful interface is a challenge.

A fundamental difference between these two philosophies is how they treat data. In Unix, files are treated merely as byte streams; they have no structure or type. Files are commonly passed from one filter to another, with each filter able to do whatever it wants to the original file. Commands are executed and files are passed as arguments. In contrast, the object-oriented model surrounds or protects data with a number of routines that have sole access to the data. Other processes that wish to read or modify the data have to ask the privileged routines to do so. The data is an object, and the task to be performed is in a sense an argument.

A different problem involves some of the strengths of Unix: I/O redirection, pipes, and background execution. How can they be accurately represented by the interface in a clear and straightforward manner? How does one graphically represent a pipe? Implementation notwithstanding, how can these features be supported while remaining true to an interface we've defined? The immediate solution was to avoid these strengths, at least at the command level, while each feature can still be used from within programs.

3 Using the Object Shell

3.1 Goals

Before embarking on this project, we wanted to set some goals to reach. Though they were only qualitative, they were at least factors to keep in mind during the design and development of the interface.

Appear Graphical & Object-Oriented

We wanted to present a graphical object-oriented interface, in hopes that a "traditional" interface such as the Unix command line interpreter could be made more approachable. As Budd's paper described, each Unix file would be represented by an icon, with pop-up menus providing the valid messages for each type of file. Dialogs and the ability to select icons as arguments added to the graphical nature.

The object-oriented features mentioned earlier are essential to our interface definition. We hope that by using them we can provide benefits for people who only use the existing classes as well as those that want to build upon it.

Appear Consistent

In general, it is important to provide consistent behavior in a user interface. A user will become accustomed to the behavior of an interface, and eventually will rely on this consistency. This will lower the learning curve when the interface is consistent across applications. However, it is important to note that there are occasions when consistency can be sacrificed. Occasionally, a break with a paradigm is more natural for a user than to live and die by a set of interface rules. Unfortunately, when to make this tradeoff is not something that can be derived mathematically; the answer lies in the results of empirical studies that reflect the whims and biases of users, a group whose behavior is not always predictable.

For example, the designers of the Macintosh interface broke with their paradigm on at least one occasion. Generally, dragging an icon over the trash can represents deleting that object. However, when you drag a floppy disk icon over the trash can, it ejects the disk. This is an inconsistency, but most people feel comfortable with the deviation from the model.

Cater to a Diverse User Group

Another goal of any user interface is to be "friendly" to a diverse group. It should be intuitive for new users. In our interface, someone shouldn't have to know that *cc* is how the C compiler is invoked, or that *-P* is the printer option to route a job to a non-default printer.

An interface should also be efficient for veteran users who don't want to resort to the time consuming steps that a new user might. A regular user, or a previously novice user who has learned over time,

should not be slowed down by an interface. There should be shortcuts available for users that are more familiar with a system.

It is understandably difficult to reach both of these goals all the time. Providing some sort of shortcut is not always easy, nor does it necessarily fit within the model. And though offering a command line might make an advanced user very happy, it isn't necessarily the best way to provide shortcuts.

Support the Strengths of Unix

This was an experiment in mapping Unix functionality to another paradigm, so we wanted to preserve as many of the strengths of Unix as possible. The three main strengths we hoped to support were I/O redirection, pipes, and background execution.

Pipes and I/O redirection are two facilities that are often used to create larger applications out of the smaller tools or filters. Background execution lets users perform multiple tasks asynchronously.

These features contribute to the differences between the object-oriented and Unix philosophies. However, they are such an integral part of Unix that we will try to include them in our model.

3.2 The Interface

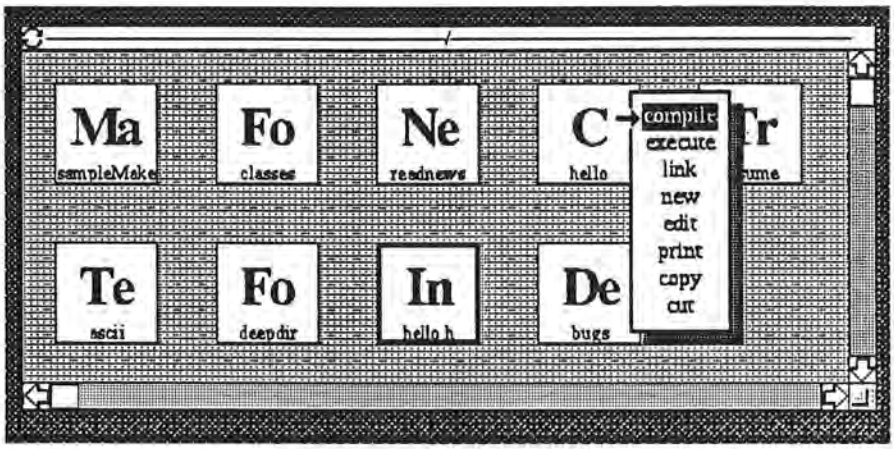

Figure 1 The OSH root folder

What the User Sees

Earlier we saw that Budd defined a framework from which to build our interface, but further definition was beyond the scope of his paper. This paper will pick up where he left off, as we look at the graphical and object-oriented features of our interface in more detail.

Each Unix file is an object, and appears as an icon. The most common exception to this is a Unix directory (called a *folder* in our interface), which can also be represented as a window, with the contents of the directory displayed as icons in the window. When the object shell is started, a window will display the root of the OSH file system, and the title bar of the folder will show the path of the OSH root directory. When a folder is opened, it creates a new window that is offset slightly from the window that contains the closed representation of that folder.

Each icon contains two pieces of information. The name of the object is in small text at the bottom of the icon. The first two characters of the object's class are displayed in a large bold font as the icon's image.

Figure 1 shows the OSH root folder opened with its contents represented as icons. The menu shows commands that are valid for a C object. The **Include** object has been highlighted and will be passed as a positional argument when the *compile* message is sent.

Mouse Behavior

Because the object shell was developed on a Sun workstation, we followed the default actions for mouse button behavior [Sun87]. The left button is called the *PointButton*. For our purposes it is used to select or unselect objects used as arguments. The middle button is the *AdjustButton*, and it can be used to move windows. Usually it is also used to move icons, but that has been disabled in the object shell. The right button is the *MenuButton*, and it will bring up a context-specific pop-up menu when it is depressed. By *context-specific* we mean that you see a different menu depending upon the class of the object that the cursor is over when the *MenuButton* is pressed. By doing this the new user will see what operations are valid for a particular object. There is no need to remember command names.

Object-Oriented Design

Every Unix file is represented as an object, and is therefore in the **Object** class or some descendent class of **Object**. (We will always list a class name in bold type.) A *descendent* class of **Object** is some subclass of **Object**, where the class can have **Object** as its immediate superclass, or it can have classes in between them. All other classes are descendents of the **Object** class. Likewise, an *ancestor* class is either the immediate superclass of a class, or some superclass thereof. **Object** has no ancestor class, and every other class has **Object** as an ancestor class.

The folders that represent Unix directories are of class **Folder**, and are a subclass of **Object**. So a folder responds not only to all messages in its own class, but also to all messages defined in class **Object**. When the mouse is pressed over a window background, you see the menu for the class of the window, which is **Folder** or some descendent class of **Folder**. You see the same menu whether the folder is closed (as an icon) or open (as a window), though not all messages make sense in both cases.

The context-specific menus described above display the names of the messages that can be sent to an object. You often have access to messages in multiple classes, starting with the messages from the class of the object at the top of the menu, working down to all its ancestor classes at the bottom of the menu. Therefore, all objects have the messages *copy* and *cut* from class **Object** at the bottom of the menu. Overriding messages are shown only once in the menu.

The methods in the object shell use inheritance to cut down on multiple copies of code. For example, class **Textual** is a subclass of **Object**. When you want to create a new text object, the *new* method in class **Textual** invokes the *new* method in class **Object** to complete the tasks necessary for all new objects. Then the *new* method in class **Textual** will perform tasks specific to text objects. This capability will be discussed in more detail in the section *Programming in the Object Shell*.

Figure 2 shows an example of code reuse. In the top window, the *new* message is sent to the instance *ShScript* of type **Class**. (The italic and bold print are supposed to help clarify the difference between the name and the class of an instance, respectively.) The *new* method in **Class** invokes *new* in **ShScript** since the *new* message was sent to the class instance *ShScript*. The *new* method in class **ShScript** invokes *new* in its superclass, **Textual**, which in turn invokes *new* in **Object**. The *new* method in class **Object** is responsible for putting up the dialog in the middle window. Once a name for the object has been entered and the OK button has been pressed, the three *new* methods terminate in the opposite order in which they were invoked, and a new instance of class **ShScript** (noticeable by the big **Sh** in the icon) will exist. By using code in its ancestor classes, the *new* method in class **ShScript** did not have to explicitly do any of the work necessary for new instances of **Textual** or **Object**.

The Object Shell Menu

Most Unix commands that take a file as an argument fit naturally into the object-oriented model: treat the file as an object, and treat the command as a message that is to be sent to the object. But Unix commands without arguments do not have a clean representation in the object shell. It is for those commands that we created an *object shell menu*, which does not simulate the sending of a message to an object. The OSH menu is located in the frame menu, which provides window movement and resizing capabilities to all windows in NeWS. It appears at the top of the frame menu and is a *walking menu*, so if you keep the mouse button depressed and move slightly to the right of the *OSH Menu* menu item, a sub-menu will appear. A picture of the OSH menu can be found with the discussion of I/O redirection.

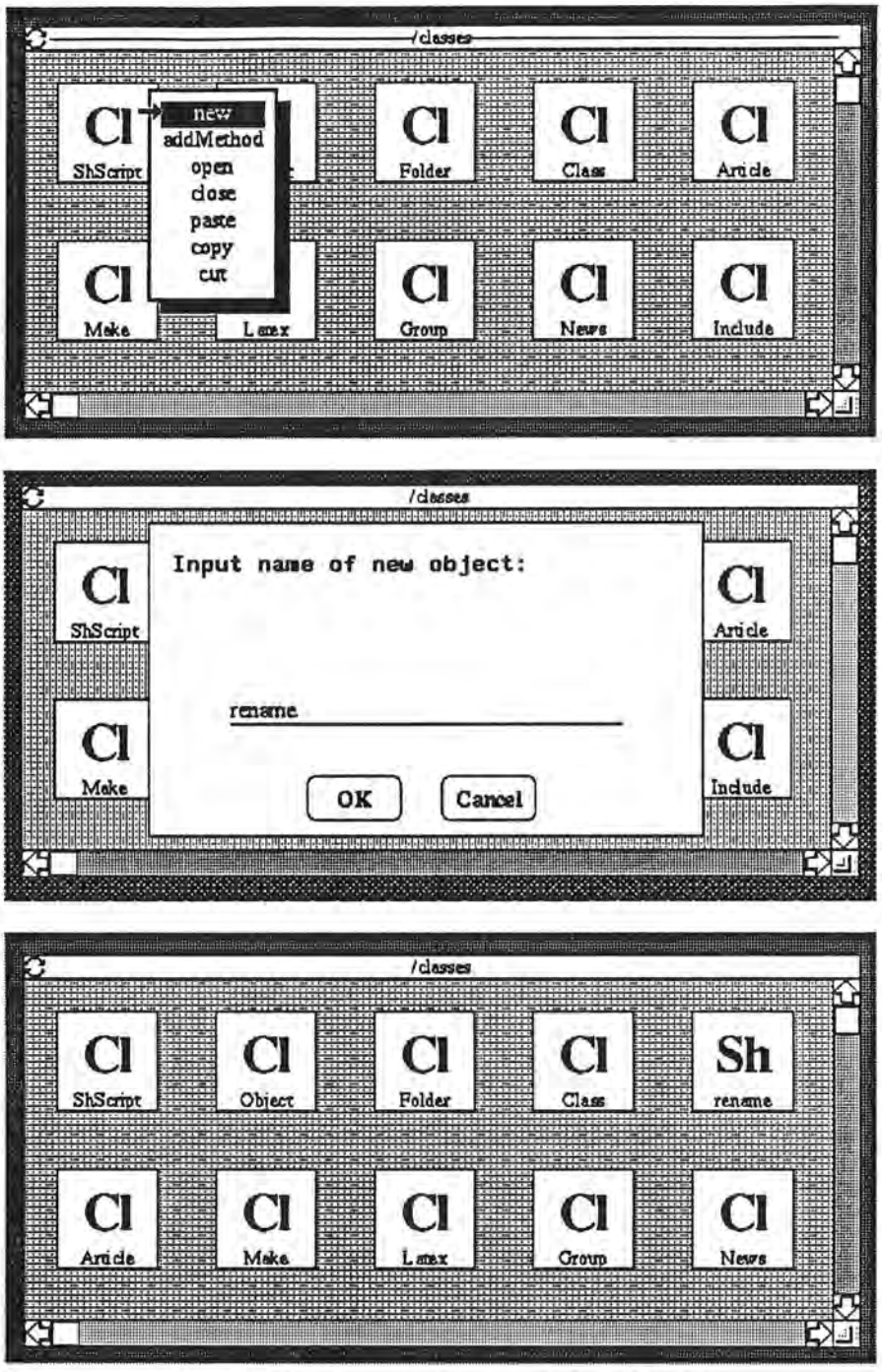

Figure 2 Creating a new instance of class **ShScript**

The OSH menu currently provides only for I/O redirection and creation of a console. Descriptions of these operations can be found later in the paper. In a more complete implementation the OSH menu might include items to list the active processes or to see who else is logged in.

The Object Shell Console

While trying to construct our graphical interface, we realized that existing Unix programs rely on *stdin*, *stdout*, and *stderr* for communication with the user. Any I/O exchanges between a user and a program

or any messages produced by a Unix tool involve these file descriptors. So the object shell must have a way to handle this textual input and output.

We chose to do so with an *object shell console*. Like the OSH menu, the console does not fit into the object-oriented paradigm. But it is a feature that is necessary in order to make the object shell functional with existing tools. The object shell console can be opened by choosing the *console* menu item in the OSH menu. Once the console exists, it is the default source and destination of *stdin*, *stdout*, and *stderr*. If no console exists, all three file descriptors are set to */dev/null*.

The Object Shell Buffer

Since we chose the cut and paste approach to deleting objects or moving objects around, there needed to be a buffer that would hold the affected object. This buffer is not visible to the user, but it saves a single object that is sent the *cut* or *copy* message. Sending the *paste* message to a folder retrieves the object residing in the object shell buffer and places it in that folder.

3.3 Class Descriptions

A base set of classes has been defined and implemented in order to give the user wishing to extend the object shell something to build upon. Where noted, a class has been defined but not implemented if another class demonstrating similar function has been implemented.

C

The class C represents all behavior that a C source file or executable program can have. It is a subclass of **Textual**, so it can be printed or edited. Once you have typed in a C program, send *compile* to the object to compile it. Likewise, *link* generates an executable instance variable. Finally, sending *execute* executes the compiled version of the C source code, using the OSH console for default input and output.

When you compile a C object, you can pass **Include** objects by selecting them before you send the *compile* message. Likewise, you can include a C library (represented by a C object) by selecting the C object before sending *link* to the C object that references the library.

Class

This class provides most of the extensibility in the object shell. All classes must reside in the */classes* folder of the object shell. The object shell expects to find the classes and accompanying methods here, and it will react unpredictably otherwise. Its superclass is **Object**, and it has two methods. The first is *new*, and by sending it to an instance of type **Class**, you can instantiate that class as shown in Figure 2. When you send *new* to a **Class** instance, it asks you to enter the name of the new instance it is creating, and then creates it in that folder. To create a new class, send *new* to the object named *Class*. It will ask you to type in the name of the new class, which should begin with an upper-case letter. It will then display a dialog asking you for the superclass of the new class.

There is a shortcut for users familiar with creating new classes. Before sending the *new* message, select the object that is the superclass using the *PointButton*. Then the selected object will be passed as an argument of the *new* message, and you will be prompted only for the name of the new object.

The other method in this class is called *addMethod*. It is used to add a new method to an existing class. To add a new method to a class, first create an object that will become the method. Currently, only objects of class **C** or **ShScript** can be methods because they are the only classes that have an *execute* message. We informally consider these to be of an executable type, though they are formally both subclasses of **Textual**. Type in the code that will make up the method. Chances are, it will be difficult to debug the method by sending it an *execute* message, especially if it needs to do anything elaborate like requesting services from the OSH server (discussed later). To make debugging easier, you can add the new object to the proper class and then test it. To do so, send *addMethod* to the class you want to add the method to. You will be prompted for the full OSH path name leading to the new method.

A shortcut can be used here as well. Select the new method and then send *addMethod* to the object of type **Class** that you want to add the new method to.

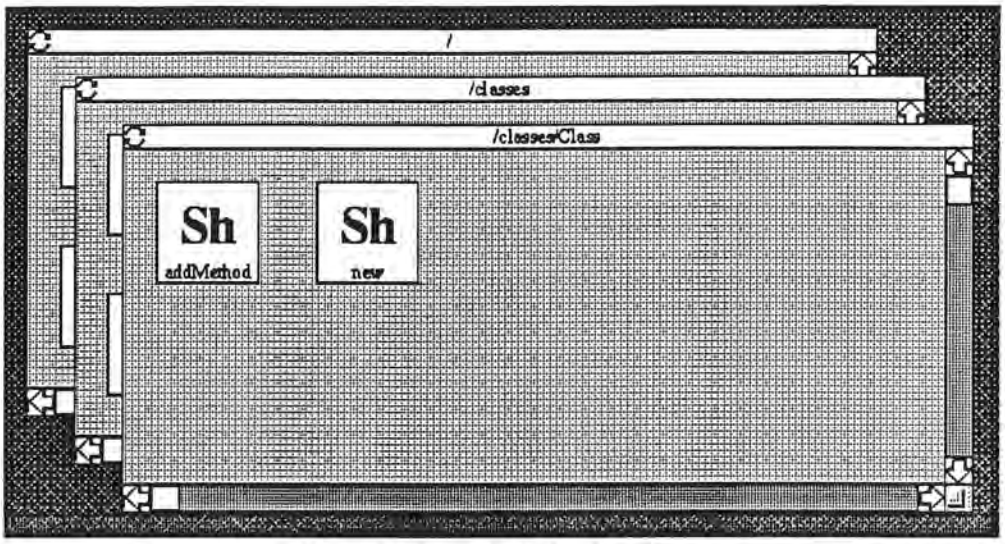

Figure 3 The methods in class **Class**

Regardless of the process you used to add the new method, all existing instances of that class, as well as all instances of descendent classes, will now have an additional message in their menu (assuming the new method is not overriding an existing method in an ancestor class).

To look at the methods for a particular class, open the */classes* folder and send *open* to one of the instances. You will see a window with all the method objects for that class. Since a method must respond to an *execute* message, all methods are in class **C** or **ShScript**. Figure 3 shows all the method objects in class **Class**.

Folder

As we've mentioned numerous times, a folder is the equivalent of a Unix directory. It contains objects, including other folders. It retains the hierarchical structure of the Unix file system. Class **Folder** is a subclass of **Object**, and the class contains three methods. Messages can be sent to folders in either an open (window) or closed (icon) state. To bring up the menu in an open folder, press the *MenuButton* while the cursor is over the window background. Sending *open* to a folder displays the contents of that folder. Sending *close* removes a window from the screen, symbolically closing that folder (and removing the window). It does not make sense to open an already open folder, or send *close* to a folder that is in iconic (closed) form. Finally, sending *paste* to a folder places the current contents of the OSH buffer into that folder. The *cut* and *copy* methods in class **Object** can be used to place objects into the buffer.

Include

Class **Include** is a subclass of **Textual**, and an instance of class **Include** is equivalent to a C header file. This class does not introduce any new methods, though the most common operations that might be performed are the *print* and *edit* methods inherited from class **Textual**.

Mail

The **Mail** class is used to represent Unix mail. It is possible for multiple instances of class **Mail** to exist, as each instance represents a mailbox. A user generally has one **Mail** instance that corresponds to the active mailbox, and others can exist that correspond to mailboxes where old mail is saved. The class **Mail** is a subclass of **Folder**, so instances respond to *open*, and *close*. It also responds to a *compose* message for sending mail. Opening an instance of **Mail** displays instances of class **Letter**.

The class **Letter** represents individual Unix *mail* messages, and are a subclass of **Folder**. They are never instantiated by the user, since the mail object handles the creation of new letters when *compose* is sent to the **Mail** object. Objects of class **Letter** respond to mail messages as one would expect. Sending

open to the object opens a window and displays the letter, and *close* likewise closes the letter and deletes it. The *reply* message allows the user to respond to the sender of that letter. Sending *save* will save the letter in the Unix default file *mbox* before deleting it from the active mailbox.

There was enough similarity between the **Mail** and **News** classes that we chose to implement only one of them. And since windowed mailers are rather common, we chose to concentrate on a windowed *readnews* tool instead.

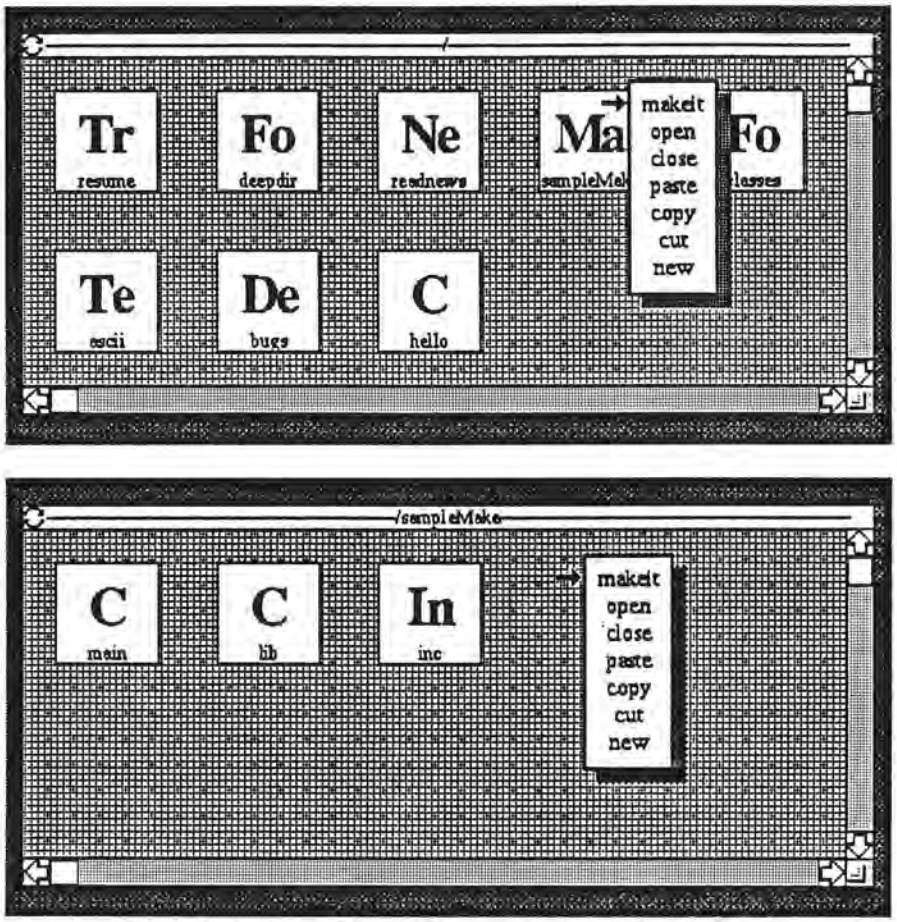

Figure 4 A sample **Make** folder in iconic and window forms

Make

The *make* facility under Unix lets you automate the management of large numbers of files with interdependencies. Probably the most common use of *make* is to define compilation dependencies between routines that access one another. This capability is one that we wanted in the object shell as well.

The **Make** class is a subclass of **Folder**, so instances respond to *open* and *close* the same way folders do. You can create a new instance of class **Make** the same way you create instances of other classes, by going to the */classes* folder and sending the message *new* to the class **Make**. The only types of objects that are allowed in a **Make** folder are:

- Objects of class **Include**. These are the objects described earlier that are equivalent to the C header files found in Unix.
- Objects of class C. One of these objects must be called *main* and will become the entry point for the compiled and linked executable. To include objects of class **Include** into your C objects, put a line in the C object like

```
#include "fred"
```

if you have an **Include** object called *fred* in the **Make** folder.

Once all the objects of class C and **Include** are created, send the message *makeit* to the **Make** folder. This can be done either from within the **Make** folder by bringing up a menu over the folder background, or by closing the folder and pressing the *MenuButton* while the cursor is over the icon of the **Make** object. Once the *makeit* process has completed, the resulting executable object will be the C object named *main* in the **Make** folder.

Figure 4 shows a **Make** folder in closed and open form. The lower window shows the open folder and all the objects that will be used to build the *main* object when *makeit* is sent to it. Note that the menus in both windows are the same, though not all messages are valid in both forms. Sending *close* to the **Make** object in the upper window would not make sense.

This is admittedly a limited implementation of *make* that supports only simple dependencies. One could imagine a more sophisticated method called something like *depend*. By selecting objects *A*, *B*, and *C*, and then sending *depend* to object *D*, we could generate two lines in a *makefile* describing a dependency where routine *D* must be recompiled if any of the other routines change. Repeating this process a few times with different objects and dependencies could result in a *makefile* that has more complicated dependencies, as well as removing the requirement that a C object named *main* exist in the folder.

News

This class implements the USENET *readnews* facility. It is a subclass of **Folder**, and thus responds to *open* and *close*. When you send *open* to a **News** object, you will see a folder containing instances of class **Group**, one for each newsgroup with unread articles. The **News** folder should contain only objects of class **Group**. Likewise, each object of class **Group** is a folder that can contain only objects of class **Article**, corresponding to individual news articles in that group.

To view a certain group with unread articles, send *open* to the object of class **Group**. If you want to look at an unsubscribed group or a group with no unread articles, send *get* to the **News** object. It will prompt you for the name of the group and make sure it is a valid newsgroup before adding it.

Group is a subclass of **Folder**, so objects of class **Group** respond to the same messages as folders, as well as two others. *Catchup* will mark all articles in the group as read, and *unsubscribe* removes the **Group** object and no longer show it when it has unread articles. *Readnews* articles are represented as instances of class **Article**, and are a subclass of **Folder**. The **Article** class does not add any methods of its own.

One might argue that **Article** should be a subclass of **Textual**. That argument has some validity. The *print* message from class **Textual** would make sense if **Article** were a subclass of **Textual**, but the *edit* message would not. And since the *open* and *close* messages from **Folder** were used to represent entering and leaving instances of both **News** and **Group**, it was decided to follow this consistency instead of any relationship that an **Article** instance may have with all text objects.

Figures 5 and 6 show how some of the **News** methods work. Once *open* is sent to the instance of class **News**, you see the top window in Figure 5, showing all the groups with unread articles. Sending *open* to a **Group** object displays the middle window with instances of class **Article**. Finally, sending *open* to an article displays the actual article.

Once the **News** object has been opened, you can send *get* to the **News** object to retrieve an unsubscribed group or a group with no unread articles. The middle window in Figure 6 shows the dialog where you type in the newsgroup of preference, and the lower window has displayed the group.

Object

The **Object** class is at the top of the class hierarchy. It has no superclass. When modeling the Unix file system, it becomes an abstract superclass; there are no instances of class **Object** in the object shell. But the class does provide methods that apply to every object in the object shell. The two methods provided by the **Object** class are similar to those found on the Macintosh: *cut* and *copy*. When you send either of these messages to an object, the object is placed in the OSH buffer. The contents of the buffer can be retrieved using the *paste* message in class **Folder**. The difference between the two messages is that *copy* leaves the receiving object alone, whereas *cut* deletes the object from the folder.

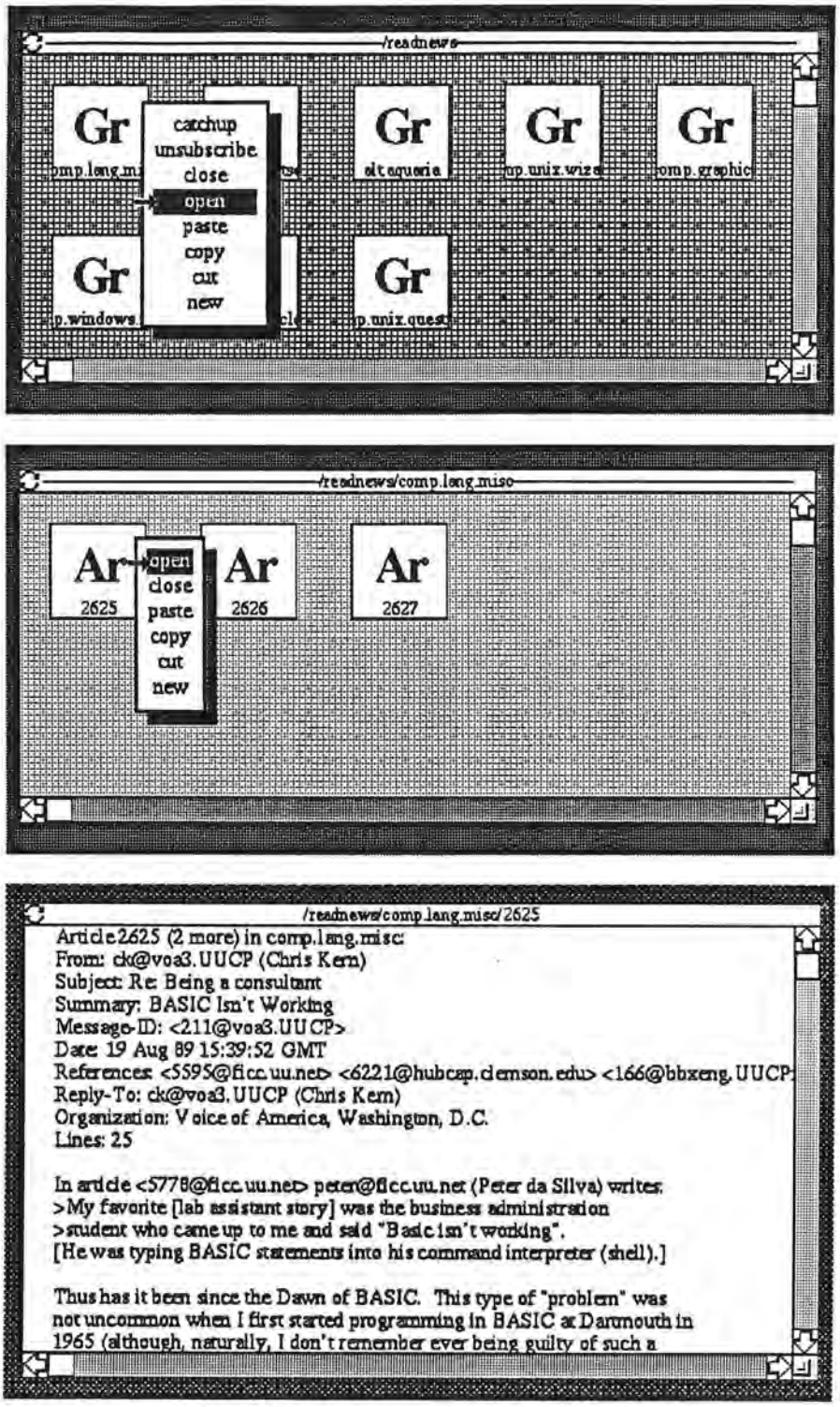

Figure 5 Steps to view an instance of **Article**

ShScript

Instances of this class are the equivalent of shell scripts in Unix, and the class is a subclass of **Textual**. The object shell accepts any of three script types: Bourne shell, C shell, or PostScript shell. Which interpreter to invoke is specified on the first line of the shell script. Besides the messages it inherits from classes **Textual** and **Object**, instances of this class also respond to *execute*, which will execute the object.

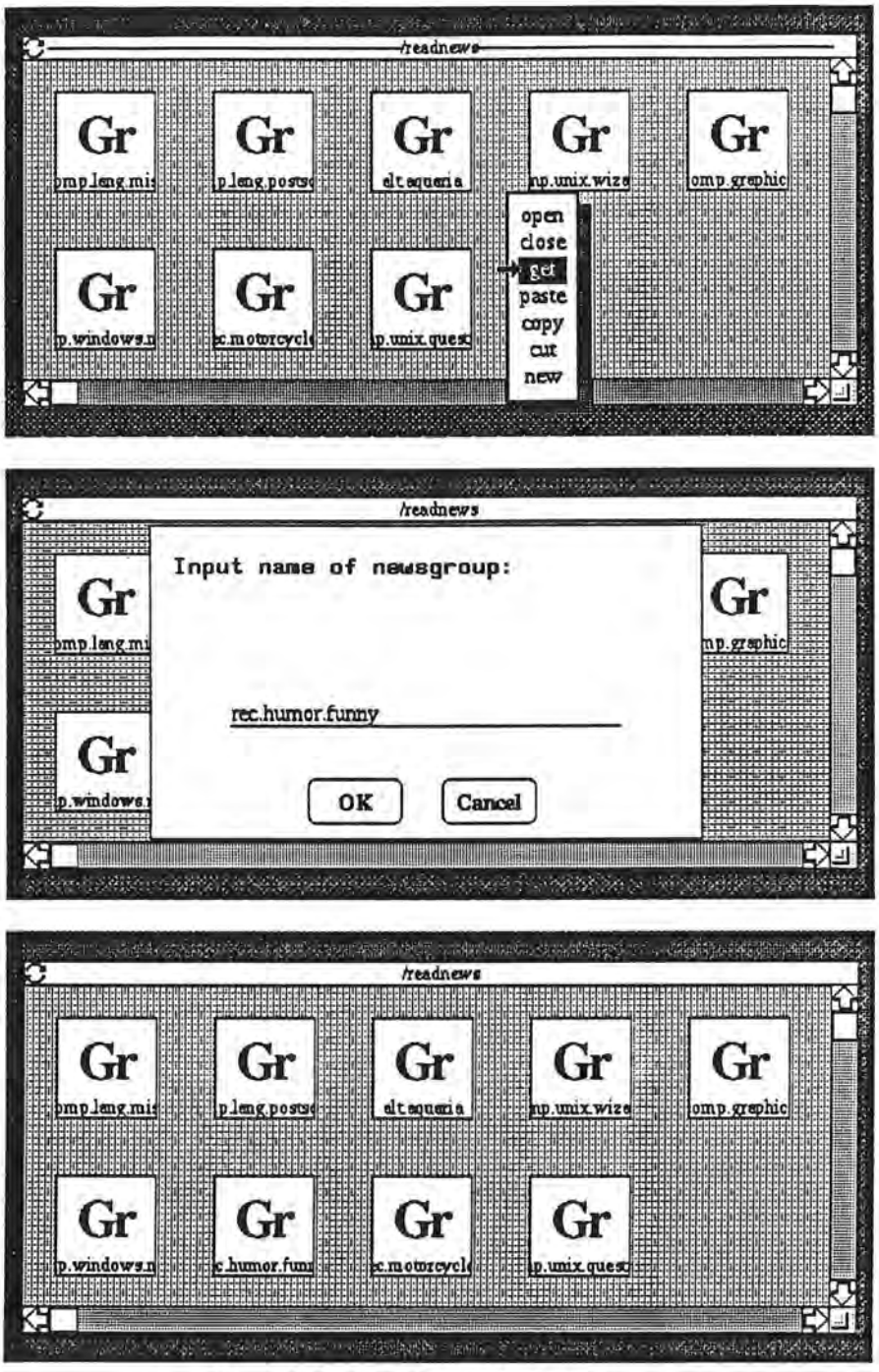

Figure 6 Getting an undisplayed newsgroup

The PostScript shell script is something unique to the NeWS environment. To invoke the PostScript interpreter, the script should contain

```
#! /usr/NeWS/bin/psh
```

on the first line. Note that due to the current OSH implementation, a PostScript shell script will not be able to access any of the OSH data structures located in the object shell kernel. This will be discussed further in the *Implementation* section. All referenced routines and data must be either self-contained or defined in the global PostScript dictionary *systemdict*.

Textual

Most Unix files will be represented within the object shell as instances of class **Textual** or some more specific descendent class. The **Textual** class is a subclass of **Object**, and it contributes two methods to the object shell. They are *edit*, which will bring up the *vi* editor for that particular object, and *print*, which will send the text representation of the object to the default line printer.

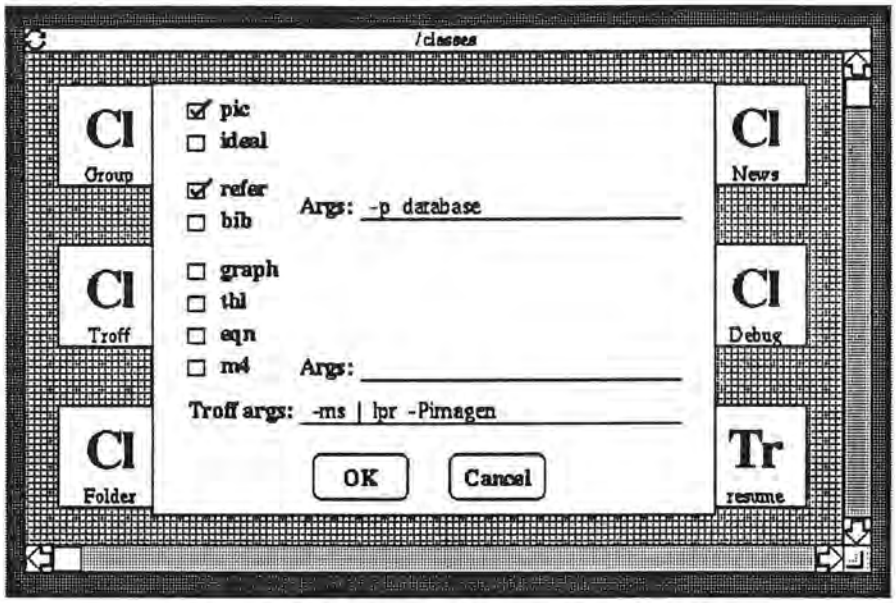

Figure 7 Dialog for new instances of class **Troff**

Troff

The **Troff** class represents *troff* documents. It is a subclass of **Textual**, and has a message called *format* that will prepare the object for printing and send it to an output device.

In the Unix environment it is common to use other filters (such as *refer* or *tbl*) in combination with *troff*. We support that capability in the object shell. When a new instance of class **Troff** is created, a dialog (shown in Figure 7) is displayed that gives the user a chance to specify various filters to include when the object is formatted. There are also fields to type in options that will be passed to the filters and *troff* itself. Once the OK button is pressed, a string is generated that contains the information entered into the dialog. (See the *Implementation* section for the string generated by this dialog.) Thus, each **Troff** instance can have unique properties. One common *troff* option may be to redirect the output to a printer that is not the default, which can be specified on the *troff* line in the dialog.

3.4 Programming in the Object Shell

Useful Library Routines

The object shell kernel (described in the *Implementation* section) acts as a resource server for all methods. It can provide NeWS services such as updating a window on the screen, and it can provide class services such as supplying the class of a particular object. To make new method creation as easy as possible, we designed a few routines that hide some of the complexity of our implementation.

All service requests are initiated in methods by a call to *sendCommand* which identifies the desired service to the OSH server. Any required arguments are passed using the *sendArgs* routine. If the OSH server returns a reply, the reply can be read using *getArgs*.

We attempt to use inheritance whenever possible to reduce redundant code in the object shell. When passed an object and a message, routine *send* will find the first class where that message is defined, and send that message to the object. When passed similar information, routine *super* will also send a message to the object, but it will begin looking for the message in the superclass of the specified class.

Without *send* and *super*, methods would need to be hard-coded with the class of methods they invoke. For example, when you instantiate the **ShScript** class, the *new* method in class **ShScript** calls *new* in its superclass, **Textual**, which in turn calls *new* in its superclass, **Object**. Without *super*, the method in class **ShScript** would need to have the call to the **Textual** *new* method hard-coded. Then later, if for some reason we chose to remove the **Textual** *new* method, we would have to modify the method in class **ShScript** to reference *new* in class **Object** instead of **Textual**. Having to remember these special cases is prone to error. But if we use *super*, then the *new* method in class **ShScript** will dynamically find the first ancestor class that contains *new*, whether it be in class **Textual** or class **Object**.

A handful of small utility routines rounds out our library: the *dirname* routine is used to return all but the last level of a path name. Though it was described in the SunOS manual pages, the routine could not be found, so we wrote our own. And the routines *gettmp* and *puttmp* are used to simulate passing information between methods.

A Method Walkthrough

To give an example of what a method looks like, we describe a sample that shows some of the features of the object shell and the library routines. Figure 8 shows the Bourne shell script for the *new* method in class **Object**, and Figure 9 shows the shell script for method *new* in class **Textual**. First we'll walk through the method for class **Object**.

The comments on each line with *sendCommand* are the names of the C constants for the request. C routines can reference these constants and though scripts cannot, they have been added as a form of documentation.

When the user selects a menu item and effectively sends a message to an object, the object shell will execute a method and pass it the object as the first argument. Methods are also passed integers representing the file descriptors used for communication with the OSH server, as well as any arguments needed by the method. Briefly, the *new* method in class **Object** asks the user for the name of the new object, checks to see if an object with that name already exists, and if not, creates an object. Now we will examine the method in more detail.

Line 5 exists in any method that wants to access the library routines; it adds the library directory to the search path. Lines 8–17 make sure that the object is of type **Class**. Instances of a class should not be able to instantiate the class. Lines 20–22 put up a dialog asking for the name of the new object, and then read the response[1]. If the returned value is a null string (meaning the user clicked the Cancel button instead of the OK button), then exit. We'll explain line 26 later. Line 34 tests to see if an object with this name already exists. If so, put up a dialog stating so and exit. If not, do what's necessary to create a new object: first, create a directory with the new name. In that directory, create an ASCII file that contains the class of this new object. Tell the OSH server to add a new icon in the window (lines 42–43), and then

[1] Any C routines that include *$OSHHOME/src/osh.h* have access to server constants. In this case, we are using the constant `#define DLOGR -3`.

```
1    #! /bin/sh
2    # Class: Object    Method: new
3    # $1=objectPath $2=writePipe readPipe $3=args
4
5    PATH="$OSHHOME/lib:$PATH"
6
7    # Make sure this is not an instance!!
8    sendCommand "$2" "53"            # GETCLASS (objPath) => class
9    sendArgs "$2" "$1"
10   class=`getArgs "$2"`
11   if [ "$class" != "Class" ]
12      then
13         sendCommand "$2" "1"       # DLOG (msg)
14         sendArgs "$2" "You can only send 'new' to a Class object."
15         puttmp ""
16         exit
17   fi
18
19   # Ask user for name of new object
20   sendCommand "$2" "-3"            # DLOGR (prompt) => reply
21   sendArgs "$2" "Input name of new object:"
22   newobj=`getArgs "$2"`
23   # Check for canceled method
24   if [ "$newobj" = "" ]
25      then
26         puttmp ""
27         exit
28   fi
29
30   # Make sure an object doesn't already have that name
31   objpath=`dirname $1`
32   class=`basename $1`
33   cd $objpath
34   if ` test -s $newobj `
35      then
36         sendCommand "$2" "1"       # DLOG (msg)
37         sendArgs "$2" "An object of that name already exists."
38      else
39         mkdir $newobj
40         echo $class > $newobj/.class
41         # Add icon to the screen
42         sendCommand "$2" "7"       # ADD_OBJS (name,class)
43         sendArgs "$2" "$newobj,$class"
44         # Redraw the window
45         sendCommand "$2" "5"       # REDRAW
46         # Save the name in case this was invoked w/ super
47         puttmp "$newobj"
48   fi
49
50   exit
```

Figure 8 Listing of method *new* in class **Object**

redraw the window (line 45). The call to *puttmp* on lines 15, 26, and 47 will be explained next when we talk about the **Textual** method.

```
1    #! /bin/sh
2    # Class: Textual   Method: new
3    # $1=objectPath $2=writePipe readPipe $3=args
4
5    PATH="$OSHHOME/lib:$PATH"
6
7    # Execute 'new' for superclass of Textual
8    super "$1" "$2" "Textual" "new" "$3"
9    # Get name that was tucked away
10   objname=`gettmp`
11   if [ "$objname" != "" ]
12      then
13         objpath=`dirname $1`
14         cd $objpath/$objname
15         touch text
16   fi
17
18   exit
```

Figure 9 Listing of method *new* in class **Textual**

The *new* method in class **Textual** is relatively small and straightforward, thanks in large part to the *new* method in class **Object** that it inherits. Line 8 in Figure 9 is a call to the *super* routine in our library. Since text objects are also a type of object, we can use the *new* method in class **Object** to first create the object. The arguments passed to *super* are: the object, the file descriptors, the class of this method which we want the superclass of, the message to look for beginning in the superclass, and any arguments passed.

Line 10 shows how we use the *puttmp* and *gettmp* routines to simulate communication between method calls. In an object-oriented system like Smalltalk, methods return an object by default. In this case, the *new* method would return the new instance (or a nullstring if the user aborted). We simulate this by placing the object we want to return into a private buffer. The method in class **Object** places the name of the new object (or a nullstring) into the buffer using *puttmp*, and the method in class **Textual** can read it after control has returned on line 10.

On line 11 we can see if the user aborted during execution of the **Object** *new* method. If it wasn't aborted, we create an instance variable *text* (in the directory representing the object) that all instances of class **Textual** have. It is this instance variable that is referenced when the *print* or *edit* message is sent to the instance.

Creating a New Method

We can create a new method and add it to the object shell very easily. We will show how to add a method called *rename* to the **Object** class. We choose to make this a Bourne shell script, because we might need to communicate with the OSH server, and a C-shell script cannot utilize pipes the way we need to.

To create the method, first go into the */classes* folder and send *new* to the class object named **ShScript**. When prompted, give it the name *rename* which will create an instance of class **ShScript**. Now send *edit* to the object, and type in the code. The code in Figure 10 will do what we want. Since we can't change the name of an existing icon, lines 29–32 delete the old icon and add a new one with a new name and the old class.

Once we have finished editing, we want to make a method out of this **ShScript** instance. Click the *PointButton* over the object to select it. Then send *addMethod* to the **Class** instance named *Object*. You are now done! The message *rename* has been added to the menu for every instance of class **Object** (and every descendent class of **Object** as well), and the method was copied into the **Object** class. (Since the superclass of **Class** is **Folder**, you can send *open* to *Object* to see all the methods in class **Object**, including the one you just added.)

Notice that the new method we created does not take care of renaming all objects. Some classes have an instance variable that is the name of the object. (See the *Implementation* section for a list of all

```
1    #! /bin/sh
2    # Class: Object    Method: rename
3    # $1=objectPath $2=writePipe readPipe $3=args
4
5    PATH="$OSHHOME/lib:$PATH"
6
7    # Ask for new name
8    sendCommand "$2" "-3"           # DLOGR (prompt) => reply
9    sendArgs "$2" "Input new name of object:"
10   newname=`getArgs "$2"`
11
12   # Check for bailout
13   if [ "$newname" = "" ]
14      then
15         exit
16   fi
17
18   # See if one already exists
19   objpath=`dirname $1`
20   objname=`basename $1`
21   cd $objpath
22   if ` test -s $newname `
23      then
24         sendCommand "$2" "1"      # DLOG (msg)
25         sendArgs "$2" "Sorry, $newname already exists."
26      else
27         mv $objname $newname
28         class=`cat $newname/.class`
29         sendCommand "$2" "10"     # DEL_OBJ (obj)
30         sendArgs "$2" "$objname"
31         sendCommand "$2" "7"      # ADD_OBJS (name,class)
32         sendArgs "$2" "$newname,$class"
33         sendCommand "$2" "5"      # REDRAW
34   fi
35
36   exit
```

Figure 10 Possible code for new method *rename* in class **Object**

classes and their instance variables.) All this method does is rename the object. It can't be responsible for checking all descendent classes for instance variables with the same name. So in order to fully implement *rename* correctly, you would have to create methods for the classes that have these instance variables. Within those methods you could use the *super* library routine to call the *rename* method in class **Object** to perform the work common to all objects.

3.5 Support of Unix Features

As we mentioned before, Budd's initial paper decided not to deal with certain aspects of Unix for the prototype. It was felt that I/O redirection, pipes, and background execution were difficult enough problems *at the command level* to delay their consideration. They are still supported within programs or scripts that may be executed by the object shell. Though most were not implemented, we at least looked at these factors and tried to visualize how we might represent them within an object-oriented paradigm.

Input/Output Redirection

There were a number of ways to represent the redirection of input and output. Using the technique we implemented, the OSH menu has two items called *inputFrom* and *outputTo*. By default, I/O will be

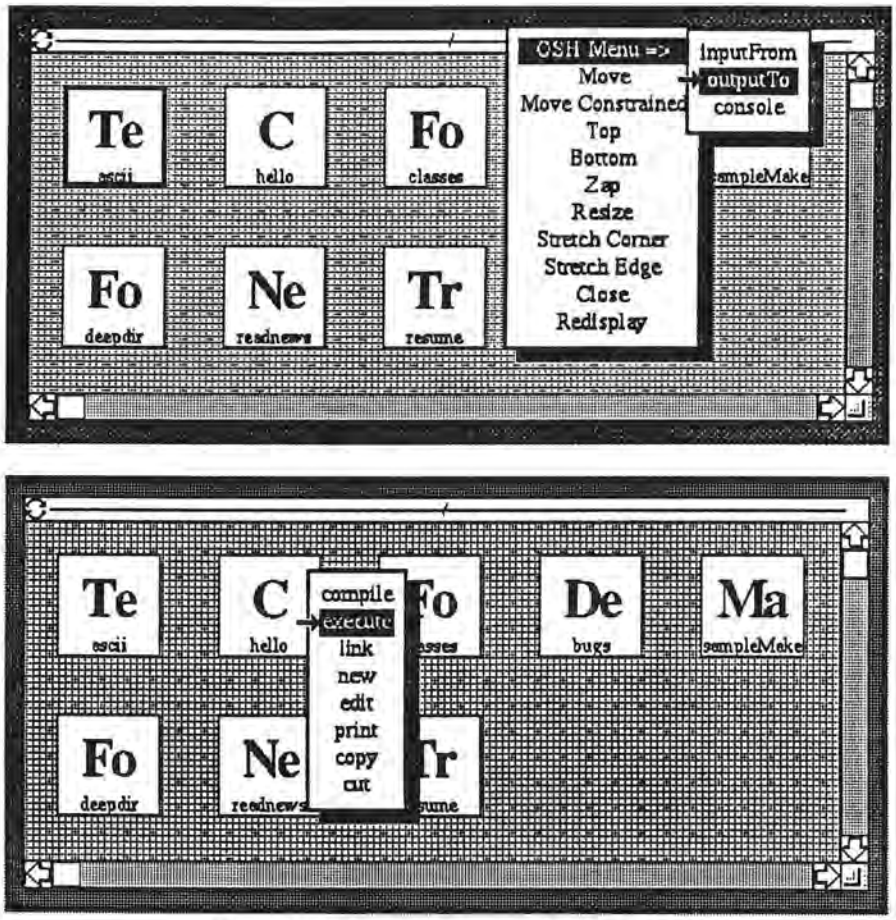

Figure 11 Redirecting output of a C object

directed to and from the OSH console, which takes the place of *stdin* and *stdout* in the Unix shell. However, input or output can be redirected on a command-by-command basis (that is, redirection only takes effect for one message send). For example, to execute a C object and redirect output to a text file, first select the desired text file and choose the *outputTo* menu item on the object shell menu. Next, send *execute* to the desired C object. The output replaces any existing text in the **Textual** object. Similar results can be achieved by redirecting input.

Figure 11 shows how this process works. In the upper window, the OSH menu is found by "walking" right along the first item in the frame menu. Selecting *outputTo* while the *ascii* textual object is highlighted sets up output redirection for the next message sent. Recall that the OSH menu does *not* generate a message send. In the lower window, *execute* is being sent to a C object called *hello* that prints the string "hello world". After the message has been sent, the string will have replaced the contents of *ascii*.

Remember that after a message has been sent, both input and output are reset to the OSH console. If you have specified redirection and then decide against it, you can undo the redirection by having no object selected when the *outputTo* or *inputFrom* menu item is chosen.

It is not possible to redirect output to a text object that doesn't exist yet (not unlike the Macintosh), because in the object shell you can reference only objects that you can select or send a message to. Therefore, the object must exist in order to be referenced.

There is another way to redirect I/O that is possibly more intuitive than the former methods, but that requires the use of graphics. This technique would use a feature where holding down the *PointButton* and dragging the mouse would create a rubber band line until the button is released, at which point a line would be drawn with an arrow at the second endpoint. This could be used for both input and output redirection, depending upon the types of objects at either end. If the arrow pointed from a C object towards a **Textual** object, then output would be redirected. If the arrow pointed from a **Textual** object towards the object to be

executed, input would be redirected. This form of redirection could be undone by dragging the arrow out of the window. As with the former techniques, the redirection would only take effect for one command.

The graphical approach may allow output redirection into an object that does not yet exist. By having the arrowhead point to the background of the window where the output object will reside, we could specify where the new object would go. In the implementation, the kernel could detect the need for a new object and invoke *new* in class **Textual** before the original method was executed. This way, the invoked method would never have to know that a new object was created for this command. But in the interface, does it make sense to have an arrow point to nothing? Even though this is unambiguous, is it a reasonable way to graphically represent the implicit instantiation of an object?

Pipes

With pipes as well as with redirection, there are a number of possible representations in the object shell. A less graphical solution would involve a meta key in combination with the mouse. By sending a number of messages to objects while a meta key is depressed, the object shell would build a sequence of objects and messages connected by pipes but not yet executed. Once the meta key is released and an item from the OSH menu called something like *doPipe* is selected, the object shell would then begin execution of the objects in the proper order, with intermediate I/O piped from one executable object to the next in the sequence. I/O redirection could be included in this example without ambiguity.

We have seen how graphical arrows could represent I/O redirection. If an arrow connected two executable objects instead, one could imagine how a pipe might work. An arrow could lead from an object generating output to an object that requires input. Unfortunately, this cannot be done without ambiguity. If you have an arrow pointing from a **C** object to a **ShScript** object, it could either represent output redirection to a text object or a pipe to an object that will be executed.

Ours is not the only attempt at integrating pipes and I/O redirection into an interface for Unix. The Open Look design team has not been able to define a satisfactory graphical representation for pipes and filters either [SuO89]. If nothing else, this is convincing evidence that the problem of an accurate representation is not a trivial one.

Background Execution

We weren't able to come up with a satisfactory way to represent background execution. This is a result of our limited model where objects represent Unix files. If we were to have a **Process** class where an object could represent a running process, we would need methods like *kill*, *suspend*, *foreground* and *background*. We would also want some way to get process information, similar to the Unix *jobs* command.

If such a class existed, **Process** would be a descendent class of **Object**, which means it would respond to the *copy* and *cut* messages. Would we equate *cut* with a Unix *kill()* and *copy* with a Unix *fork()*? How would we store such an object in the OSH buffer? It is clear that a whole new set of possibilities and problems arise when we let an object represent something other than a Unix file.

3.6 Comments

Shortcuts

We found that our interface does not provide shortcuts for advanced users, or for users that learn with time and want to skip the steps that they used while learning. This is unfortunate. We could have added a command line to appease Unix hackers, but that would have been a quick fix and not something that was carefully thought out to fit into our model.

Probably the most glaring example of the lack of shortcuts was the *readnews* tool. Granted, the command line version of *rn* was designed to be efficient and to require a minimum of keystrokes. In the object shell, even a minimum of menu clicks takes longer and could eventually become aggravating for regular users. In our defense, let's not overlook how a new user might react to the two different *readnews* interfaces. The OSH version would almost surely be easier to use.

Supporting Different User Groups

Though we found that the object shell does not provide many shortcuts for advanced users, it can be a productive environment for these users nonetheless.

While occasional users may be satisfied with the classes and methods that already exist to perform their tasks, advanced users and hackers might want to dabble at the Unix level. Developers could create applications in the Unix environment with system calls galore, which could then be invoked by methods in the object shell. The News class is an example of this. Other users not as familiar with Unix internals could define new classes or methods to take advantage of existing Unix tools, similar to our **Troff** class.

This ability to satisfy different user groups was suggested by Budd [Bud89], and our further experimentation supports his observation.

Limitations

Due to the complexity of NeWS and our unfamiliarity with it, some things were not done simply because we did not know how. And once we learned how, we were far enough along in our implementation that time did not permit the necessary changes to be made.

For example, all user actions are a result of menu selecting. There is no direct icon manipulation such as drawing arrows from one icon to another to represent redirection or pipes. Also, some operations are clumsy. Moving an object involves cut and paste operations. In comparison with the Macintosh Finder, where icons can be dragged from one folder to another, our method is indeed awkward considering how often the operation is likely to be performed.

Still, considering the expressive restrictions of our interface, we were able to demonstrate that common tasks can be performed without too much effort.

Large Menus

As more classes and more methods are added to the object shell, the size of menus will increase. At what point has a menu grown too large, and how could we break large menus up? If we broke them up based on classes and provided a walking menu with the primary menu showing the classes, then we would lose a sense of polymorphism in our system. The user would need to know what classes have certain messages, and in what classes messages are overridden. Another technique might be to break the items up into levels according to frequency of use. This is subjective, and not something that a designer or user could easily and accurately predict.

Factory Methods

The design of the object shell was heavily influenced by the structure Brad Cox uses in his book [Cox86]. His definition of a class includes two types of methods. The *factory* methods are those that apply to the entire class, or that might set a class variable that will apply to the entire class. For example, the *new* method that creates a new instance of a class should really only apply to the class itself. An *instance* method is one that can be sent to a specific instance of the class. Sending a *rename* message to an instance renames only that instance. Note that class definitions are themselves instances of class **Class**. For example, the *Troff* class object (that defines class **Troff**) also responds to the *rename* message.

After working with both types of methods, we came to the conclusion that factory methods were a mixed blessing. They were useful in that if methods did arise that made sense only when applied to the entire class, we could support them. An unfortunate side effect is that a user wishing to add a method to a class needs to understand the difference between the two types of methods, and has to specify whether the method being added is a factory or instance method. We felt this was unacceptable. Even users well versed in object-oriented design might not be familiar with Cox's terminology.

In the end we chose not to offer two types of methods, which has caused one unfortunate quirk in the object shell. When factory methods were supported, there were two menus for each class: one for factory methods and one for instance methods. But now a single menu contains all valid messages for a class. Therefore, every object in the system now contains the *new* message in its menu of valid messages, because there is a *new* method in class **Object**. When sent a *new* message, instances will display an error

dialog, since only objects of type **Class** should respond to *new*. On the positive side, we have not come up with any other needs for factory methods. If we had chosen to implement the **Troff** class differently, we might have wanted a **Troff** class variable that would be set using a factory method. This could be done if we wanted to have a **Troff** class variable, such as the printer where the *troff* output would be sent.

Adding New Methods

In order to provide shortcuts or options for users who prefer doing things a certain way over another, we tried to avoid long sequences of operations like answering many dialogs. For instance, when a user creates a new class, he is prompted by the object shell for the name of the new class and its superclass. The user may speed up this process a little by selecting the superclass before sending the *new* message to **Class**. Then the user is prompted only for the name of the new class, and the *new* method reads the superclass from the list of selected icons.

We tried to offer the same sort of flexibility when adding a new method to a class, but were only partially successful. The beginner's way to add a new method to a class is to send *addMethod* to a class instance, at which point a dialog will ask you to type in the full OSH path to that method. Implementation problems kept us from a more elegant solution. Instead of prompting for the full path, the object shell could have displayed a dialog telling the user to select the method to be added, and then have the user click the OK button on the dialog once it was found. This may have required moving around the object shell and opening folders if the method was not already visible, and it was an instance when concurrency would have been useful. In its current form, the object shell can only have one method executing at a time. A method must terminate before another message can be sent to an object. If *addMethod* displayed a dialog prompting the user to find the method to be added, *addMethod* would wait until the user had found the object and selected it. But opening a folder requires the *open* method, and it would not execute until *addMethod* had terminated.

Pure or Practical?

There is a fine line between straying from the object-oriented paradigm and providing functions that do not follow the paradigm but are added to make the use of the object shell easier. In other words, it is acceptable to provide something that doesn't exactly follow the model if there is no straightforward way to do it from within the model. But if we can do something within the object-oriented model, there is no reason to add another way to do it that is outside the model.

This was the philosophy we took with the object shell. We were willing to go to reasonable lengths to keep our interface consistent. In extreme cases, this attitude can lead to a product that is pure but useless because of its awkwardness. We tried to be patient and resist quick solutions that were not consistent with our interface behavior.

One example of this is the arrows that were proposed for I/O redirection. Using arrows would make the process more graphical, but it would not agree with our admittedly minimalistic graphical interface. Another feature that could have been added was the movement of objects by dragging them à la Macintosh Finder, which would have been much easier than the current method of cutting and pasting.

Textual vs. Executable

The object shell supports single inheritance only. This is not to say that there weren't any occasions where multiple inheritance might have been useful. Take the classes **C** and **ShScript** for example. Both are a subclass of **Textual**, which makes sense since both have a textual representation, and both will need to be printed or edited on occasion. But both of these classes could also be a subclass of **Executable**, if such an abstract superclass existed. We could use this to specify that all methods must have **Executable** as an ancestor class. A redirection arrow would then connect an **Executable** instance (or descendent) with a **Textual** instance (or descendent). An arrow that was being used as a pipe would connect two instances with an ancestor class of **Executable**.

Budd envisioned an OSH menu item called *execute* that would execute a selected icon [Bud89]. Having an **Executable** class would have made that easier. We have chosen to give the textual relationship priority over any other similarities. Providing an *execute* item in the object shell menu would demonstrate how

an imperative execution capability could be combined with the existing paradigm. However, it would not enhance the object-oriented model we are trying to develop, and it is not as appropriate without an **Executable** class.

Representation of Objects

In our design we chose to represent objects by the standard size icon. It seemed to be a natural shape to use; there was a label along the bottom for the object's name, and given the time, we could have designed a more elaborate image than the first two characters of the object's class.

By the time we started to define the classes for *mail* and *readnews*, we realized that the icon was going to be too restrictive a representation; the label of an icon can be on the average ten characters long. This became a problem for three of our classes: **Group**, **Article**, and **Letter**. Newsgroup names would need to be either truncated or abbreviated in order to fit most of them into ten characters. An article now displays the article number in the icon, which is unique but not very informative. We would almost have to show the subject line *and* the article number since subject lines aren't necessarily unique. And though we didn't implement Unix *mail*, we did realize that we would have a similar problem with **Letter** instances. The letter number would be unique but uninformative, so we would most likely need at least parts of the header line in order to be unique.

One solution to this problem would be to make the size of the icon adjustable. If the label were 40 or 80 characters, we could have much more descriptive object names, though we still could not guarantee uniqueness.

While defining the *mail* interface we thought of another potential problem. If a user opens up his **Mail** object and sees a group of uniquely named objects, he still won't know which of the **Letter** instances have been read and which haven't. A more flexible object representation would allow the icon drawing to change depending upon the state of the object. Using Sun's *mailtool* as an example, when there is no unread mail, the *mailtool* icon displays a mailbox with its flag down. When new mail arrives, the flag goes up.

4 The Implementation

The object shell was developed on a Sun-3 workstation running SunOS 3.5. It was implemented using a combination of C, NeWS, and shell scripts. The total lines of code were approximately 1000 lines each of C, NeWS and shell scripts. The methods were written mostly using the Bourne shell, except for a couple methods that were written in C. The object shell kernel was written in C and NeWS.

The Network Extensible Window System is a network-based windowing system by Sun Microsystems. It is a superset of Adobe Systems' PostScript that includes primitives for network communications and event handling. Abstract data typing and message sending are part of the extensions to PostScript, and the NeWS system comes with a basic set of class definitions for windows, menus, and dialog boxes.

The *cps* facility of NeWS allows communication between C programs and the NeWS address space. By compiling a file that contains PostScript routines with C-like procedure headers, you can invoke PostScript routines from C and pass data between the two environments.

The object shell consists of four parts: The driver (or main routine) spawns a daemon process and then acts as the main event loop for the life of the object shell. The OSH server is a daemon that satisfies NeWS and class requests from methods. As the driver and daemon are the core of the object shell and are not meant to be modified, together they are called the OSH kernel. The OSH file system is an abstract hierarchical file system in which all objects reside. Finally, the classes and methods created from within the OSH environment actually define the state and behavior of all objects in the OSH file system.

Figure 12 shows how the pieces fit together. The object shell server is the interface between methods and both the NeWS environment and the class mechanism. In most respects, the main routine is treated just like a method. The OSH server cannot tell the difference between a request from the main routine and a request from a method. Note that C-shell methods have no access to the OSH server, so they are very limited in what they can do. PostScript methods can access the global NeWS environment directly, but not the class structure nor the PostScript routines defined by the OSH server.

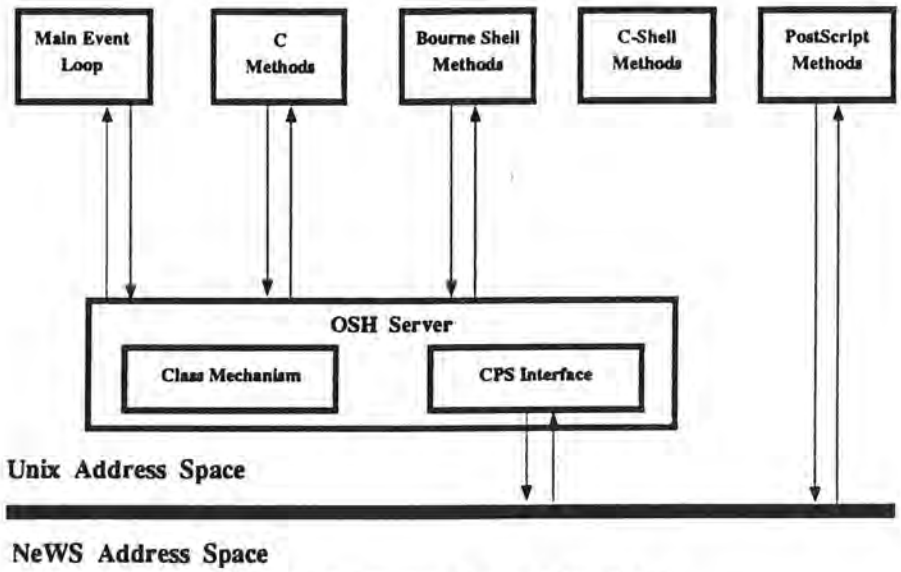

Figure 12 Structure of the Object Shell

4.1 The Object Shell Driver

This routine is the entry point of the object shell. Upon invocation, the driver creates two pipes: one to be used by methods to send requests to the OSH server, and another for the server's replies. It then forks off a child process that becomes the object shell server. Since a child process inherits the open file descriptors of its parent, the daemon process will have both pipes needed for method-server communication.

Once the OSH server has been started, the driver goes into a main event loop. The loop continually waits for menu input from the user. It sends a *read-menu* request to the server and waits until the server returns the result of a menu click. The returned information will include the object and the message sent to it, as well as any message arguments[2]. The driver does some minor processing of the returned information, and then returns the data and requests a *messager* event, which asks the OSH server to carry out the operation on the passed object. The driver cannot carry out the operation itself, because only the class mechanism in the OSH server knows the relationships between classes and where to look for methods. Once a *messager* request is sent to the daemon, the main event loop will go into a synchronous wait until the method terminates. The driver will continue in this loop until the server responds to a *read-menu* request with a flag stating that the user wishes to exit. At this point the driver closes all open pipes and terminates.

4.2 The Object Shell Server

The object shell server controls access to the class mechanism and the NeWS environment. Since scripts or independently compiled programs cannot directly access either the class structure or NeWS environment, the server communicates with methods (and the main event loop) via Unix pipes. When the daemon process is initiated, it inherits open file descriptors from the main routine which will have created the necessary two pipes

The most common requests made to the server are *read-menu* and *messager* requests. When a *read-menu* request is received, the server will pass control to a PostScript routine to detect a menu item selection. Once a selection has occurred, NeWS will return an object, a message (the menu item selected), and a list of any other objects that were highlighted when the menu item was selected. The server then writes this data into the pipe read by methods. Soon the server will get a *messager* request with the same data (albeit slightly modified), and pass the information to the messager routine.

For another example of what the object shell server does, consider the *new* method in class **Object**, which will need to create a new icon in the current window to represent the new instance. After the

[2] Message arguments in the object shell are equivalent to keyword arguments in Smalltalk, where *index* and *value* are the keyword arguments in the statement `myArray at: index put: value`.

method *new* has created a new object in the Unix file system, it will need to send requests to the NeWS interface (via the OSH server) to create a new icon, re-arrange the icons in the window, and redraw the window. Or, if a user wanted to add a new method to the **ShScript** class, the method called *addMethod* would use the OSH server to update the class data structure to reflect the addition of a new message to the class **ShScript**. The server would also be asked to update the NeWS menus for the **ShScript** class and any descendent classes.

A different approach to distributing tasks between the server and methods could have been taken. The kernel could be responsible for knowing all the steps that need to be taken for a certain operation, so that methods don't have to. This would make methods smaller but the kernel much more complex. For instance, in the first example above, the method could simply tell the OSH server the name of a new instance of a certain class, and make the server remember the correct sequence of steps to update the screen and the class data structure. Doing so would increase the chances that the kernel would have to be modified when new methods are added.

The object shell server attempts to supply all the necessary functions that a method might need. These are relatively low-level functions so that the kernel will not have to be modified to support new methods. Hopefully, any services a new method might need can be obtained using a combination of the existing services. This approach was taken in order to make the environment as extensible as possible. Standish has claimed that many extensibility experiments have failed because the systems were so complex that they resisted change [Sta75]. We feel that by keeping requests simple, more can be done in the object shell without modifying the OSH kernel.

It is for this reason that a window isn't automatically redrawn if an icon is added or deleted. An explicit *redraw* request must be made since a method may add multiple icons, and we don't want the window redrawn until the last icon has been added.

The Class Mechanism

The class mechanism consists of a data structure containing all class information and the routines used to access or update that structure. For every class, a part of the structure holds the superclass of the class and all the messages that an instance of the class should respond to. When the OSH server first begins, it calls a class initialization routine, passing the Unix path to the root of the OSH file system. The initialization routine will expect to find a folder called */classes* where all the class definitions and methods reside. It will scan all classes and load class names, superclasses, and method names into the data structure used for message lookup at run-time. If modifications are made to a class definition during execution, the changes will be made both to the OSH file system and the internal data structure. As a result, any changes to the class definitions are reflected in persistent memory (the Unix file system), and will remain between OSH invocations.

When a method makes a class related request, the object shell server calls one of a number of routines that only it has access to. Typical operations that the server may satisfy are: finding the class of an object, returning a list of messages that a class will respond to, and adding a new class to the class data structure.

The most frequently used routine in the class mechanism is the *messager*. When passed an object, a message, and any message arguments, the messager will find the first class where the message exists in the class data structure. If the message doesn't exist in the class of the object, the messager will check the superclass for it, and continue to follow the superclass chain until the message is found. Once the message is found, the messager knows where to find the method in the OSH file system, since all class definitions (and the corresponding methods) must exist in the */classes* folder. Using the *fork()* and *execl()* system calls, the messager will spawn a new process and overlay the child with the proper method. The newly spawned method will inherit both open pipes, and will have the resources to communicate with the OSH server in order to make NeWS or class requests.

4.3 The Class Methods

It is the methods that give the object shell its power and extensibility. As we noted above, we designed the OSH server so it would supply a base set of functions that we thought would satisfy all requests from

methods. The OSH kernel should never need to be modified once a comprehensive set of NeWS and class services have been defined and implemented.

Any object that is executable can be a method. Currently, C and **ShScript** are the only implemented classes that have an *execute* message. Instances of class **ShScript** can be written for either the Bourne shell, C shell, or PostScript shell by setting the first line of the script accordingly. However, only the Bourne shell can be used if a method plans to communicate with the OSH server, because it is the only shell that can redirect I/O to a pipe. Most methods implemented so far have used the Bourne shell for this reason, with the exception of the *open* and *close* methods in class **Folder**, which are C objects in order to speed up opening and closing of folders.

When a method first gains control after being spawned by the messager, it is passed three arguments: the object that this method shall operate on, the file descriptors used for communication with the OSH server, and any message arguments that may be passed. The arguments will vary depending upon the method being invoked and whether or not the user selected other objects before sending a message to this object. Currently, only the *makeit* method in class **Make** takes advantage of message arguments. It passes a list of **Include** objects when *compile* is sent to each of the C objects, and it passes a list of the compiled C objects when it sends *link* to the C object named *main* in the **Make** folder.

Though the Bourne shell is preferable, either the C shell or the PostScript shell can be used to develop new scripts that will become methods. The C shell might be used by someone who wants a more C-like syntax, while the PostScript shell would allow direct manipulation of the screen. But only methods using the Bourne shell will be able to use the library routines developed to ease new method creation.

4.4 The Object Shell File System

When the object shell is started, it looks for the *$OSHHOME* environment variable, which specifies where the object shell directory resides. The subdirectories of *$OSHHOME* we need are *bin* (containing binaries for the main routine and the daemon), *lib* (containing the library routines), *root* (the root of the OSH file system), and *src* (containing the PostScript code that is loaded at run-time).

In the underlying implementation, each object is represented by a Unix directory. When a class is instantiated, a Unix directory is created with the name of the object. The directory will have an ASCII file called *.class* that contains the class of the object. Consider this file the sole instance variable for class **Object**. Directories that represent an instance of **Class** (a class definition) will also have an ASCII file called *.super* that contains the superclass of the class. The *Class Implementation* section lists the methods and instance variables for all implemented classes.

4.5 Class Implementation

We have tried to define a base set of classes that are appropriate for a simulation of the Unix environment. Classes and methods have been defined that will allow for common operations to be performed. We have listed the classes alphabetically here.

In the following descriptions, *self* is the object that is receiving the message. This is similar to the way Smalltalk identifies the receiver. It is also the name of the directory that represents the object and contains any instance variables of the object in the Unix file system.

Article

Superclass:	**Textual**	
Instance Variables:	none	
Methods:	*close*	Mark article *self* as read and close the window.
	open	Create a window and read article *self*.

Instances of class **Article** represent articles in the *readnews* application. It is a class that can only be manipulated while the application is running, and there is no Unix directory for each instance. Dummy methods exist in the OSH file system so that the messages are added to the **Article** class menu when the object shell begins.

C

Superclass:	**Textual**	
Instance Variables:	*self*	A symbolic link to *text* (inherited from **Textual**).
Methods:	*compile*	Compile the source file *self*.c into *self*.o.
	execute	Execute the binary file *self*.
	link	Link the object file *self*.o with any passed libraries, creating *self*.
	new	Invoke *new* in **Textual**, and create the *self* instance variable upon return.

An object of class C is similar to an ASCII file with the *.c* suffix. The C code itself resides in *text*, but the symbolic link was added since the C compiler prefers file names with a suffix. The executable is called *self* instead of *a.out* so that either a C or **ShScript** object can be executed by specifying *self*.

Class

Superclass:	**Object**	
Instance Variables:	*.super*	An ASCII file containing the name of the superclass.
Methods:	*addMethod*	Add a method to class *self*.
	new	Create a new instance of class *self* by invoking the *new* method in class *self*.

There is no equivalent to the **Class** type in Unix, since Unix is void of any type information. This class forces a type, at least upon objects in the OSH file system. As with all methods that move objects around, *addMethod* uses *tar* to move the new method in order to preserve any symbolic links. Since the *new* method is invoked every time a class is instantiated, it determines the class of *self*, and then starts looking in that class for a *new* method.

Folder

Superclass:	**Object**	
Instance Variables:	none	
Methods:	*close*	Close folder *self* and remove its window.
	open	Open folder *self* and display the contents in a new window.
	paste	Copy the contents of the object shell buffer into *self*.

An object of class **Folder** is the equivalent of a Unix directory. Both *open* and *close* are written in C for speed, although they could easily be Bourne shell scripts.

Group

Superclass:	**Folder**	
Instance Variables:	none	
Methods:	*catchup*	Mark all articles in group *self* as read.
	close	Leave newsgroup *self* and remove the window.
	open	Open window and display all unread articles in newsgroup *self*.
	unsubscribe	Unsubscribe from newsgroup *self*.

The class **Group** represents a newsgroup within the *readnews* application. It is a class that can only be manipulated while the application is running, and there is no Unix directory for each instance. Dummy methods exist in the OSH file system so that the messages are added to the **Group** class menu when the object shell begins.

Include

Superclass:	**Textual**	
Instance Variables:	none	
Methods:	none	

An instance of **Include** is comparable to a C file that ends in *.h*.

Make

Superclass:	**Folder**	
Instance Variables:	none	
Methods:	*makeit*	Compile all C objects, including any **Include** objects, and link to create a binary file called *self* in object *main*.

The **Make** class provides some of the functionality of the Unix *make* facility. It is a type of folder containing all objects that will become part of the final binary file. The *makeit* method will make sure all objects are either **C** or **Include** and that one of the **C** objects is called *main*, make a list of all **Include** instances to pass when each **C** object is compiled, and finally pass a list of **C** objects (except for *main*) for linking *main*.

News

Superclass:	**Folder**	
Instance Variables:	none	
Methods:	*close*	Remove the window and exit *readnews*.
	get	Show a group with no unread articles, or subscribe to an unsubscribed newsgroup.
	open	Open a *readnews* window and display all groups with unread articles.

The class **News** represents the *readnews* application. It is the only application so far that reads the menu and has a main event loop of its own. Though it has no instance variables, it is conceivable that *.newsrc* could be considered one since it defines the state of the *readnews* object. This application was developed by starting with Larry Wall's public domain *rn* code, stripping out most of the I/O support (which is sizeable), and then inserting calls to our routines which in turn requested many services from the OSH server. The logic of the original *rn* was left largely intact.

For those familiar with the *rn* source, files were not modified if the only purpose was to comment out extraneous I/O calls. The files that were modified either needed a change in logic or required services from the object shell. Those routines are: *art.c*, *final.c*, *init.c*, *ng.c*, *rcstuff.c*, and *rn.c*. In order to add as few lines to *rn* proper as possible, we added a file called *oshrn.c* that included our interface code.

Object

Superclass:	Undefined	
Instance Variables:	*.class*	An ASCII file containing the class of *self*.
Methods:	*copy*	Copy *self* into the object shell buffer, leaving the original intact.
	cut	Copy *self* into the object shell buffer, deleting the original.
	new	Create a Unix directory called *self* to represent the object and create the *.class* instance variable that holds the class of *self*.

There is no equivalent in Unix to the **Object** class. Both methods in this class use *tar* to copy the object in order to preserve symbolic links. This is an abstract class; there should never be an instance of class **Object**.

ShScript

Superclass:	**Textual**	
Instance Variables:	*self*	A symbolic link to *text*.
Methods:	*execute*	Execute the shell script in *text*. Use line 1 to determine which interpreter to invoke.
	new	Invoke *new* in **Textual** and create the *self* instance variable upon return.

The **ShScript** class can be used to represent Bourne shell, C shell, or PostScript shell scripts. The symbolic link is included so that both **C** and **ShScript** instances can be executed by calling *self*.

Textual

Superclass:	**Object**	
Instance Variables:	*text*	The instance variable that holds the ASCII representation of *self*.
Methods:	*edit*	Edit *text* using the *vi* editor.
	print	Print *text*.
	new	Invoke *new* in **Object** and create the *text* instance variable upon return.

ASCII files under Unix that don't have a more specific class to represent them are represented as instances of class **Textual**. Instances of **Textual** (or descendent classes) are the only valid source/destination for input/output redirection.

Troff

Superclass:	**Textual**	
Instance Variables:	*command*	The ASCII command string that contains all *troff* options for *self*.
Methods:	*format*	Use *command* to format and print *text*.
	new	Invoke *new* in **Textual** and display dialog upon return.

The **Troff** class is used to represent *troff* source files in Unix. When a new **Troff** instance is created, the user is shown a dialog that lets them choose options for this instance. If the user clicks the OK button, then the *new* method will generate the instance variable *command* that will record all the options. Given the dialog in Figure 7, clicking the OK button will generate the unsophisticated though effective string

```
cat text | refer -p database | pic | troff -ms | lpr -Pimagen
```

which will be stored in the *command* instance variable. When the instance is sent *format*, it will simply execute this command and the desired output will be obtained.

4.6 Comments

A Prototype

The object shell was an experiment in a different type of user interface. Since it was a prototype, our first concern was not how polished or flashy it looked. If our aim was a polished application, our priorities would have been different.

A few features were neglected in the object shell that were not necessary but which would have been more elegant. The first was any ability for a user to create NeWS dialogs other than the basic informative (shown in Figure 13) or reply (shown in Figure 2) varieties which the OSH server supports. The *new* method in class **Troff** is a complex dialog with check boxes and multiple fields that are used to choose filters and processing options. We had hoped to come up with some sort of facility that would let users create new dialogs that contained check boxes or other widgets without going through the details that were

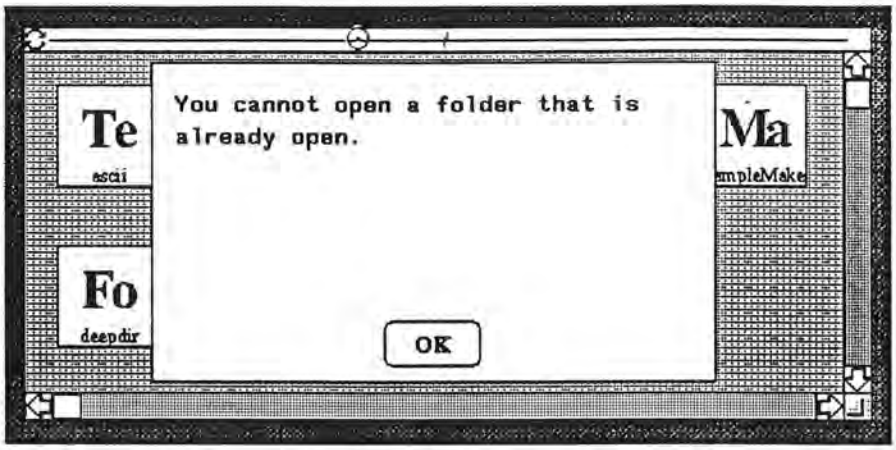

Figure 13 A simple informative dialog

necessary for the **Troff** dialog. However, automating or simplifying the construction of user interfaces is a research area in itself, and we chose not to get sidetracked.

The second feature that wasn't included was the ability to have multiple methods running concurrently. The actual simultaneous running of multiple processes would not have been difficult to implement, but we would have needed some sort of locking mechanism to protect critical objects and synchronize messages. We also would need a way to let the user know which processes were still running and which had completed. The latter problem might have been solved by having the cursor in the shape of an hourglass only when it is over objects that are "locked".

One last feature we left out involved menus that could detect the state of the object. There are occasions when some menu items for an object are not valid. For example, immediately after a new C object has been created, the *link* and *execute* messages do not make sense. Our implementation checks once the message has been sent, and if the message is inappropriate, an error dialog is displayed. We would have preferred to be able to disable menu items for those messages that are not currently valid. However, there is no existing facility in NeWS to accomplish this, and it wasn't important enough for us to devote the necessary time to develop it ourselves.

The Readnews Application

Up until the *readnews* tool was developed, a method would always terminate before the user could send the next message. The main event loop would wait until the method terminated before sending another *read-menu* request to the OSH server. *Readnews* was the first attempt at a tool that reads menu selections directly. Therefore, once *open* was sent to the **News** object, the main event loop would wait while any news is read, and would not continue until *close* was sent to the object.

A drawback to supporting a standalone application was that we had to in effect do the work of the messager for the messages we were processing. For messages that we did not process, we could use the OSH server to carry out the message send. But in our particular instance, that wasn't necessary since any message we didn't process was an invalid operation. For instance, you could send *cut* to an **Article** object, and the application would trap it and display an error message since for the *readnews* application, only messages in class **Article** can be sent to articles. We justified this as the equivalent of having an overriding *cut* method in class **Article**.

The NeWS Environment

One factor that would have led to a more polished product would have been some sort of NeWS toolkit. NeWS 1.1 does come with a limited set of classes and methods for windows, menus, and so forth, but some very primitive operations still needed to be done by hand.

As an example of the lack of basic functions, we had wanted to use a scrolling window to display text for *mail* or *readnews* articles. The supplied *ScrollWindow* class did not supply any text scrolling routines, so it would have involved writing text on a canvas, and computing where to begin each line as well as

how many characters or words could fit on a line. The methods to define scrolling would also need to be written. This seemed extreme after seeing how easily scrolling could be done using the text window class in the NeXT Application Toolkit [NeX88]. Luckily, the USENET newsgroup *comp.windows.news* has its share of NeWS wizards willing to help out, and we were able to modify some window scrolling code that already existed.

Another contributing factor was the availability of NeWS documentation. What we had was terse and had few good examples. A long-awaited book was delayed until after this project was finished.

5 What Next?

Generalize Objects

In our implementation, objects can only represent Unix files. This restriction may have kept us from representing some features of Unix. In order to take advantage of networking and concurrency, we need to have a more general object definition. That is, objects should be able to represent entities other than Unix files. In order to make the object shell more flexible, we would need class definitions to represent such objects as processes, hosts, and printers.

Work on Unix Strengths

Without a doubt, Unix derives much of its strength from I/O redirection, pipes, and background execution. With these strengths, it is possible to combine small tools or filters into a more complex tool. Unfortunately, we did not come up with any clear ways to represent these activities in our interface, and we need these features in some shape or form if the object shell is to provide the power of Unix.

The fact that all I/O in Unix is treated as character streams presents another difficulty. Ideally, an object-oriented interface would pass typed objects instead of character streams to insure that the sender and the receiver of a transaction are expecting the same type of object. This feature would add structure to the currently free-form transfer used by pipes and redirection.

More Tools

Creating (or failing to create) more tools for the OSH environment will help to prove one way or another whether the object shell concept is useful. An implementation of *mail* would be useful, and it would not be difficult to expand on current implementations of *make* or *rn*. The methods we implemented gave a general feel of the interface, but they were not full-featured or robust.

Along the lines of exploiting the strengths of Unix, once an interface definition for pipes (and thus filters) existed, implementations of filters like *grep* and *awk* would add noticeable power to the object shell.

Networking

As was mentioned by Beaudouin, not many people exist in an isolated computing environment. Machines need to talk to one another, though the user doesn't need to be aware of it. The current trend in distributed systems tends to hide networking information from the user so that he doesn't need to know if he is using a remote processor or if his files are on a remote file system. In our case, supporting a remote file system would not have affected our interface definition. Where the object actually resides is important only at the implementation level.

We might point out that although the object shell was designed for local execution only, the OSH kernel is a server that could conceivably communicate with a remote user as well as a local user if pipes were replaced with sockets.

Concurrency

Once the representation problem of background execution is solved, we can better support concurrent processes. This would allow more than one method to execute at once, which would be useful for applications like *readnews* where the *open* method in class News is essentially executing until the application has exited.

6 Conclusions

We have found that it is difficult to create an interface for a wide range of users. There need to be shortcuts for advanced users, which the object shell did not do a good job of supplying. Even casual users will improve and change over time. This deficiency was probably most noticeable in the object-oriented *readnews* tool which in its command line form is designed to be very efficient, with only a minimum of keystrokes necessary. For proficient users, it is difficult for a menu-driven system to compete with single keystrokes. But we should not forget that *rn* is not user-friendly to the new user.

We discovered that solving the problems related to I/O redirection, pipes, and background execution are formidable ones. It is clear that the implementation would be the least of our problems. Defining a concise and unambiguous representation for these features is difficult. Defining a representation that seems natural for a large group of users and that also fits within our interface model gives new meaning to the word "difficult".

Though there were snags and stumbling blocks, we don't believe they were major enough to question the usefulness of our object shell. A graphical interface is ideal for the casual user who doesn't want to do many complicated things, and the object-oriented flavor makes it even more approachable. Valid commands are easily found in an object's menu, and a user doesn't need to know how to print different types of objects. He need only tell it to print itself. Perhaps this interface cannot satisfy the needs of expert users, but since a majority of Unix users only perform a handful of operations on a regular basis [Bea89], this type of interface might prove useful.

Our classes and methods were not polished or robust, but they were complete enough to demonstrate that common Unix commands can be supported. They also demonstrated that the object shell is extensible, so that the environment can grow. The OSH server provides a relatively simple interface to services that will ease the addition of new classes and methods. It also allows development at different levels, whether the user wants to create complex Unix functions or just take advantage of existing Unix tools.

This project has answered some of our original questions, and also caused us to ask new questions that we hadn't thought of before. We have discovered that an object-based shell can accomplish much of what an average user might want to do on a daily basis. Still, there are some real limitations, or at least issues that require much more study. We must also be sure that we recognize the difference between problems with the interface model itself, and problems resulting from our implementation and understanding of some of our development tools.

Though there are still some major problems to be addressed, our results are encouraging. Graphical interfaces are growing in popularity, but no product that we know of has tried to introduce a paradigm similar to ours to the general public. Perhaps it would be too big a step for the public to take all at once. Hopefully, as new and evolving interfaces include more features of object-oriented design, enthusiasm for this kind of interface will grow and the concepts will prosper.

References

[Bea89] Beaudouin-Lafon, Michel. "User Interface Support for the Integration of Software Tools: an Iconic Model of Interaction", 1988 Proceedings of the Software Engineering Symposium on Practical Software Development Environments, reprinted as *Sigplan Notices*, February 1989.

[Bud87] Budd, T.A. *A Little Smalltalk*, Addison-Wesley, Reading, Massachusetts, 1987.

[Bud89] Budd, T.A. "The Design of an Object-Oriented Command Interpreter", *Software: Practice & Experience*, January 1989.

[Cox86] Cox, Brad J. *Object-Oriented Programming: An Evolutionary Approach*, Addison-Wesley, Reading, Massachusetts, 1986.

[Fis89] Fischer, Gerhard. "Human-Computer Interaction Software: Lessons Learned, Challenges Ahead", *IEEE Software*, January 1989.

[GoR84] Goldberg, Adele, and Robson, David. *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, Massachusetts, 1984.

[Mye88] Myers, Brad. "A Taxonomy of Window Manager User Interfaces", *IEEE Computer Graphics & Applications*, September 1988.

[NeX88] NeXT, Inc. *NeXT Application Toolkit*, 1988.

[Sta75] Standish, T.A. "Extensibility in Language Design", *Sigplan Notices*, July 1975.

[Sun86] Sun Microsystems. *Windows and Window Based Tools: Beginner's Guide*, 1986.

[Sun87] Sun Microsystems. *NeWS Manual*, 1987.

[SuO89] "Designing the Ultimate GUI: An Interview with Dr. Lin Brown", *Sun Observer*, August 1989.

[Tei84] Teitelman, Warren. "A Tour Through Cedar", *IEEE Software*, April 1984.