The Design of an Object Oriented Command Interpreter Timothy A. Budd Department of Computer Science Oregon State University Corvallis, Oregon 97331 Object oriented programming languages are noted for their ability to allow users to quickly construct large software systems. They achieve this ability by allowing the programmer to concentrate on what it is they want to do, ignoring details of how that functionality is achieved. Such characteristics should make the object oriented style of programming attractive to casual or novice computer users, indeed one of the best known object oriented programming languages, Smalltalk, was initially designed with just such novice users in mind. The Unix operating system is widely regarded as a good environment for doing software development. Nevertheless, its large number of often terse and cryptic commands make it difficult for novice or casual users to use. In this paper we argue that one solution is to create an object oriented command interpreter, or shell, between the user and the underlying system. This paper reports on the construction of one such prototype shell, and the lessons we have learned from the use of this system. *keywords:* Object oriented, command interpreters, C, Unix Shell.

Introduction In the area of programming languages the object oriented paradigm is recognized as an important aid in the construction of large and complex software systems. This recognition is due to the support that the technique provides for data abstraction, generic operations, software reusability and inheritance [.budd little smalltalk.]. By delegating responsibility for behavior to individual objects, users can often ignore unnecessary details of implementation and concentrate on design decisions at a very high level. It has been recognized that these characteristics should make object oriented programming languages attractive to novice or casual users. Indeed, the most widely known object oriented programming language, Smalltalk, was explicitly designed with novice users, even children, as an intended user community [.kay scientific american.]. It is our contention that similar benefits will accrue from the use of the object oriented method at the system command level. By binding data and operations together, rather than treating them as separate entities, the object oriented technique permits the user to concentrate on the type of task to be performed, rather than the individual steps required to perform a task. The Unix* operating system[.bourne.] has proven, over almost two decades, to be a flexible and robust software development environment. * Unix is a trademark of AT&T Bell Laboratories Nevertheless, the Unix system is frequently criticized for being terse and unfriendly to novice users. It is therefore an interesting question to consider whether object oriented techniques might be usefully applied to develop a command interpreter for Unix that retains the power of the original system while presenting a more user friendly visage. In this paper we first describe some of the features that we would like an object oriented command interpreter to possess, most important that of a graphical interface. We comment on some of the difficulties inherent in reconciling the object oriented philosophy with the imperative and functional programming techniques conventionally found in Unix. We then describe a prototype system that we have developed to investigate some of the implications of object oriented command invocation at the shell level, and outline lessons we have learned from experience with this prototype. We conclude by describing the various directions this research is now taking.

Our Idealized Command Language

A Graphical Shell

The graphical, iconic, object oriented shell that we are striving to produce will look in many ways like other icon-based systems. To many readers, the most familiar such system is the Macintosh [.inside macintosh.], although it was certainly not the first to utilize this style of interface. More recently, the Sun workstation *Organizer* has also allowed Unix directories to be displayed in iconic fashion. On the viewers screen there will be a series of overlapping windows, some containing small pictorial images, called *icons*, each icon representing an object in the users control. With the aid of a pointing device, such as a mouse, the user can select certain objects for further processing. There are one or more buttons on the mouse; the user points to an icon representing an object to be processed and presses a button to select the object. The icon for the selected object is then displayed in a different form, such as reversing the bits in the graphical image of the icon. The mouse is also used to issue commands. There are two types of commands. Some commands, so called *system* commands, can be issued at any time. Other commands, *object specific* commands, can only be issued once an object has been selected. Unlike systems such as the *Sun Organizer*, the range of object specific commands is determined by the type (the *class*) of the selected object, and will vary from object to object. To issue a command the user pushes a button on the mouse. A menu (either a pull down menu in the Macintosh style, or a pop up menu in the Smalltalk style) appears; by moving the mouse to the appropriate entry in the menu the command is selected. One important difference between the Macintosh interface and our proposed model concerns the handling of menus. On current systems the set of items to be found on the menu is usually fixed (such as is true on most Macintosh applications) or there may be only a small set of choices (such as separate menus for directories, executable files and textual files). In our system the menu items are very dynamic, depending entirely on the type of object selected as the receiver for the command. Suppose, for example, that the user has an object representing a C program on the screen. To compile the object the user would point to it, then select the command *compile* from a menu. The object would then compile itself. The user would not need to know the name of the compiler that was being used, or any flags that might be necessary, only the fact that C objects possess the ability to be compiled. In addition to specifying commands and receivers, traditional command interpreters also permit users to specify flags which alter the interpretations of commands and arguments which may accompany a command. The graphical equivalent of command line flags would be a Macintosh style dialog box. As we will argue in a subsequent section, where we present an object oriented approach to the troff typesetting system, such dialogs should be issued infrequently, preferably once; thereafter the receiving object should remember the particular options selected, and reuse them whenever necessary. Arguments can be easily provided for by allowing the user to select any number of objects, by pointing to successive icons in turn. The selected objects are remembered, and if an object-specific command is issued the set of commands is determined by the first selected object (the so-called *receiver* of the command). The remaining selected objects, after the receiver, are treated as the arguments to the command. These arguments will be made available to the executable object chosen to implement the selected command (next section).

Integrating Procedural and Object Oriented Techniques

Command invocation in a conventional Unix system can be viewed as issuing statements in a language possessing characteristics of both a procedural and functional nature. Files are like variables, and commands are the operations used to change variables. Functions (that is, programs) are called upon to create or modify the values of variables (that is, files). In addition, by using the pipe facility, functions can be composed in order to generate new functions. We wish to retain, as far as possible, both these imperative and functional features in the new command interpreter, but in addition add to them the ability to specify actions in an object oriented manner. The first step in doing so is to introduce a new type of object, called an *executable* object. An executable object performs a single, specific task; this corresponds to a binary executable or a shell script in the conventional Unix world. To cause the program to perform this task, the user selects the executable, selects any arguments to accompany it, then chooses the system message *execute* to pass to the selected object. The execute message causes the invocation of the shell script or binary file, just as under a conventional system. The functionality provided by object specific commands is ultimately achieved by means of executable objects. As we have noted, the range of object specific commands an object will respond to is determined by the class (or type) of the object. For example, objects of class C, representing C programs, respond to commands to edit, compile, link, print, and so on. When such a command is issued, the *class description* for the class is examined. Part of the class description is a collection of executable objects, one for each command that the object responds to. If an executable object is found whose name matches that of the command, the executable is started,[†] with the object on which it is acting upon being passed as the first argument. (Any further arguments the user specified will follow this first argument). † We have rather carefully noted that the *execute* message is a system command, and thus subject to special handling. To handle the *execute* message as a normal message leads to an infinite regress; the system would search for a method to handle the *execute* message, then pass *execute* to this method, which would cause a search for a method to handle the *execute* message, and so on. Notice that this scheme differs from that of the Smalltalk-80 programming language, where *methods* (the sequence of actions to be performed in response to a command) cannot exist outside of class descriptions. By making methods be merely executable objects, we permit a more flexible style of command invocation. Commands can be given in either an object oriented postfix style (by selecting an object and issuing an object-specific command), or in an imperative prefix style (by selecting an executable and providing it with arguments).

The Advantages of Object Oriented Command Invocation A major advantage of the object oriented style of programming is that it permits users to concentrate on *what* it is they want to accomplish, and not *how* it is that this task is to be performed. Individual objects themselves are responsible for knowing the how, leaving the user to work at a much higher level. This characteristic should make an object oriented command interpreter particularly attractive to novice or casual users. One aspect of the fact that conventional Unix command interpreters require the user to remember far too many details is the large number of names the user must be familiar with to make effective use of the system. The next section will describe how the object oriented shell deals with this problem.

Relieving Name Space Congestion

The typical Unix system provides the user with an exceedingly rich set of commands. On many systems there may be over 400 separate commands that the user can issue. Corresponding to each of these commands there must exist a separate file that will be processed if the command is invoked. This is accomplished either by loading and running the file if it is a binary executable or by interpreting it if it is a shell script. The user invokes a command by naming the file in which the command information resides. Note that these files are system files, and are in addition to whatever files the user may have containing data, documents or programs. Since these system files are shared between many users, it makes little sense for them to appear in the personal directory of any individual user. Even if they weren't shared, there are far too many files to appear in one directory. Thus they are spread about in a number of designated areas, or libraries. In theory the user could execute any command by prefixing the name of the directory in which the command resides with a path specification which exactly determines the file. To do this, however, the user would have to remember which files (commands) reside in which libraries. In order to eliminate the tedium of having the user explicitly name the library, the Unix command interpreter provides something called a *search path*; a list of libraries that will be searched when a command is invoked. Thus the user need only type the command name, and the system will automatically search all specified libraries until a command is found that matches the command. The object oriented approach relieves the problem of name space congestion in an entirely different manner. To start with, there are far fewer commands for the user to remember. Objects of different classes will respond to the same command in different ways; for example both Pascal objects (Pascal programs) and C objects (C programs) will respond to the message *compile* by compiling their source code. Different compilers will be invoked, different actions taken, but from the users point of view only one command is necessary. When an action is specified via a message sent to an object, an executable taken from the class description for the receiving object is initiated. The user need not, indeed will seldom, have direct access to this method. Thus the methods need not appear on a search path, although the method must, of course, be indirectly accessible through the class description. Finally, the graphical nature of the interface makes it easy for users to remind themselves of the commands objects respond to, by merely pointing to the object in question and pushing the menu button on the mouse.

Visual Shell Programming

Another technique that can be used to reduce name space congestion is a new method for shell programming. In the traditional Unix system, executable files can be created in one of two ways; binary executable files are created by compiling programs, or executables can be constructed by means of shell scripts. The latter facility permits a new operation, the composition of executables to produce new executables. Under the conventional interpretation by the Unix shell, however, this composition is performed dynamically at run time. Consider, for example, a simple shell script named combinedProgram containing a single pipe: programOne | programTwo In order for the command to be recognized, the file combinedProgram must be accessible to the user, that is, on the current search path. Since the search for an executable to bind to the identifiers programOne and programTwo is performed at run time, however, these names must also be on the users search path, even though they may never be directly invoked by the user. (More recent Unix coding conventions have held that a shell script containing such a pipe should itself set the search path. This mitigates this problem, although this convention is not universally adhered to, and the remaining problems to be described below still apply). In place of this, we hope to construct a tool for composing actual executables to produce new executables. In place of a textual shell file, the user would manipulate icons representing the executable objects on a two dimensional tablet, explicitly showing the relationships (and types) between output of one program and input to a second. Instead of manipulating names, the actual executables would be incorporated into the new object, so that the end user need not have any access to the original objects. That is, we will provide an early binding of the basic building blocks, at composition time, instead of the late run time binding currently employed in Unix shell scripts. By providing a two dimensional framework, much more flexible interconnections become possible. In fact, we hope the resulting system will be flexible enough to permit programming of simple applications directly, in a manner similar to that of the Visual Interactive Programming system on the Macintosh [.vip.]. At the topmost, graphical level the notion of path does not make much sense; objects can only be selected if they can be seen and pointed to. In a truly object oriented system all entities, including folders (directories) should be objects, and be eligible for sharing, copying, or moving. Thus, to make a new class or a new executable available to users, the new object (the class description or the executable object) must be placed in a location from which users can copy it into their own desktop. This might be provided by a special ‘‘systems’’ folder that can be opened by a system command. Since by combining actual executables to form new executables, instead of combining names, methods can be made entirely self contained, there is little reason to retain the Unix notion of path in the new system.

Problems in Implementing Objects under Unix

The Problem With Pipes

If any feature can be said to characterize the Unix system it is the structure (or, more correctly, the lack of structure) of files and the concept of pipes and filters. These ideas go hand in hand. Since any file can be viewed as merely a stream of bytes, any program that can take a file as input can potentially accept any arbitrary file. Since a file produced as output by one program has the same structure as the file expected as input by another program, it makes sense, from both a conceptual and efficiency standpoint, to connect the two programs together directly; thus, both programs can run in parallel, one consuming bytes as the other is producing them. Pipes themselves are not at odds with the object oriented philosophy. Indeed, the pipe bar can be interpreted as a message passed to an executable object, which results in the creation of a new executable (as we described in the earlier section on visual programming). Although there are some difficulties in actually implementing pipes in this manner (a topic we will return to shortly), in theory there is no problem mapping pipes onto object oriented concepts. Instead, the difficulty with pipes and redirection occurs because of the essential typelessness of Unix files. The notion of types is very weakly defined under Unix. Clearly, we can differentiate directories from files, and we can largely differentiate executable from readable files, although some files (shell scripts) are both readable and executable. When we try to impose any further type information, such as a class, on a file we see that there simply is no mechanism to do so. To see this, consider how files are created in Unix. One way would be to use a create statement in a C program, such as the following: fileDescriptor = creat("filename","w"); The arguments to the create statement convey no information about the type of information to be written to the file. We could add an additional argument to the function to indicate the class of the newly created file, but this would necessarily invalidate all existing programs that use the original form, and force us to edit (and recompile) a large body of existing software. A second way to create a file is by redirection, as in cat >filename Again, the three symbols written here do not convey any information on the class of the resulting object. This is not to say that a new syntax could not be defined that would indicate this information, but that the existing syntax is inadequate. In short we are left with a dilemma. If we want to preserve the notions of pipes and redirection, then we must develop mechanisms for conveying class information in programs and shell scripts. The alternative is to disallow redirection and pipes at the command level. We chose the latter[†]. †. The problem of *creat*ing files with no class information remains. We are working on various solutions to this problem, however, we are hampered at the moment by the representation we have selected for objects. We will address this in a later section. In a certain sense there is a virtue to be made of this necessity. The graphical, mouse-based interface we have outlined is simple and largely intuitive. It is difficult to conceive of a protocol that could be used to supply the necessary information to support standard input, standard output, and redirection that would not involve a clumsy interplay between mouse and keyboard. By eliminating these ideas, we retain a simple interface.

A Philosophical Difference between Unix and the Object Shell

There is an even more fundamental problem that lies at the heart of the philosophical difference between the object oriented way of doing things and the functional approach of Unix. Classes and objects embody a very pure notion of abstract data types; one never modifies the actual data (the local state, or memory, of an object) directly. In the object oriented technique each object is master of its own internal state. One does not *do* something to an object, rather one *asks* the object (via messages) to do something to itself. In Dan Ingalls memorable quote [.ingalls rape.]: ‘‘Instead of a bit-grinding processor raping and plundering data structures, we have a universe of well-behaved objects that courteously ask each other to carry out their various desires.’’ The Unix technique, on the other hand, is the antithesis of this approach. The normal Unix solution to a problem proceeds by keelhauling a file through a sea of filters, subjecting it to the actions of a number of different, individual, separate programs in the process of producing the final result. Each of these programs wants to modify the data (the internal state of the object), directly, while the poor object, if it can be said to exist at all, is left on the sideline, no longer in control of its destiny. The *troff* typesetting system and its related programs are a classic example of this situation. To produce a document written in troff form, some subset of the following programs may be required[.bourne.]: The *m4* macro processor, for general purpose macro replacement. The *refer* bibliographic formatting system, or its alternative *bib*. The *pic* picture drawing program, or its alternative *ideal*. The *graph* graphics drawing program. The *tbl* table formatting program. The *eqn* figure formatting program, or one of its variants (*neqn*). Finally the *troff* typesetting system. To obtain this functionality in an object oriented framework there are several options that could be used. One would be for an object of class Troff (containing, as part of its local memory, the text of the document) to, in response to a message such as *compile*, run all programs on the input text. This option is unworkable since not only are auxiliary files required for some of the tools (a bibliographic database for *refer* and *bib*, a macro definition file for *m4*), but some of the tools are mutually incompatible (*refer* and *bib*, *pic* and *ideal*). Furthermore the particular programs required may depend upon the output device on which the final result will appear (the selection of *eqn* versus *neqn*, for example). A second option would be to create a large set of classes, corresponding to reasonable subsets of commands. Thus we would have the class BibEqnTblTroff, as well as the class ReferPicEqnTroff. While this reduces the number of commands the user must recognize (each class need only implement the commands *edit*, *compile* and *print*), the user would be forced to decide before creating a document what tools would ultimately be required to process that document; this goes against the way users ofttimes work. Instead of creating a large number of classes, a large number of objects (instances of class Executable) could be created, one for each tool. The user could then process a file by passing it through each object in turn, in the same manner as is done now (but without the benefit of pipes). In fact, the argument against this approach is that it is exactly mimicing the current way Unix works, and buys us none of the advantages of abstraction, information hiding and data structuring provided by the object oriented technique. Yet another option would be to create a generic Troff class, but provide it with a large number of different messages, such as *refer*, *tbl*, and so on. This solution, however, eliminates for us one of the main advantages of the object oriented approach, namely that the user would need to remember only a small number of commands, with each object knowing how it will respond to those commands in different ways. The best solution would seem to achieve both flexibility and ease of use. The first time a document is compiled, a dialog box would be displayed permitting the user to select which of the various tools are to be used and supplying any arguments which may be necessary (such as the bibliographic database used by *refer* or *bib*). This information would be stored as part of the state for the document object. Thereafter, the message *compile* would implicitly invoke the desired filters. Thus we achieve flexibility without the use of pipes and filters which, we have argued, seem difficult to integrate into the object oriented framework.

A Prototype Object Oriented Shell In order to gain some initial hands-on experience with the object oriented paradigm and Unix, we decided to prototype a simple version of the Object Shell. At this stage our primary goals were flexibility and rapid development. Accordingly, the initial program was written in the Bourne Shell language [.bourne.], and was textually, rather than graphically oriented. Instead of a graphically displayed desktop, where individual objects were represented by icons, a textual description of the current objects (obtained by a simple *ls* on a directory) was presented to the user prior to each command. Instead of the user pointing to and selecting an object with a mouse, the intended receiver of a message would be named in Smalltalk fashion [.Smalltalk blue.] at the beginning of each message. Instead of a pop-up or pull-down menu, each command would be written following the name of the receiver. Finally, instead of a graphically oriented dialog for entering arguments, optional arguments would be named following the message. Thus commands to the object shell all have the following form: receiver message [arguments] The code for this initial version of the object shell is given in the appendix. In order to show the differences between programming using an object shell and programming using a conventional shell, we present a small sample session using the object shell. The object shell is entered by typing *osh* with no arguments. As the shell begins, and prior to each user command, a listing of the objects in the current Folder (directory) is presented. In this session there is initially only one object, named *classes*. % osh

classes

osh% We can determine what sort of object *classes* is by passing it the message *class*. (In the final graphical system, of course, the class of an object would be indicated by its icon). osh% classes class Folder The object *classes* is an instance of class *Folder*, a collection of other objects. We can determine the messages that instances of this class will respond to by using the message *commands*. osh% Folder commands add contents open

We open the folder using the message *open*, and note that we now have a new desktop; that is, a new set of objects we are discussing. osh% classes open

C	Latex	Textual	
Class	Folder	Object	ShScript

osh% Thus we see that the folder *classes* contains seven objects. These represent seven classes that the system understands. (We are following the Smalltalk convention of using names beginning with uppercase letters for classes, and names beginning with lowercase letters for other objects. This is, however, merely convention and is not enforced by the system). These seven classes can be described as follows: A super class of all classes, the class Object provides simple basic functionality common to all objects. A subclass of Object, the class Textual provides the ability to deal with textual objects; that is, objects that can be edited. A subclass of Textual, the class C provides the ability to create, compile, and link C programs. A subclass of Textual, the class Latex provides the ability to create, compile and print documents using the Latex macros [.lamport latex.] and the Tex text formatter [.texbook.]. A subclass of Object, the class Class provides the ability to create and edit new class descriptions. A subclass of Object, the class Folder provides functionality for dealing with collections of objects. A subclass of Textual, the class ShScript provides the means to create and edit Bourne Shell scripts [.bourne.]. We can create a new object by passing the message *new* to a class. The system will prompt for the name to be used to denote the newly created object. Let us try creating a new Latex program. osh% Latex new name of new object: doc

C	Latex	Textual	doc
Class	Folder	Object	ShScript

Notice that a new object named *doc* has appeared. As with *classes*, we can discover the class of doc by means of the message *class*. osh% doc class Latex Instances of class Latex are editable (by virtue of being subclasses of Textual). Passing the message *edit* to *doc* places us in a text editor, using which we can enter a simple document: \documentstyle{article} \begin{document}

This is a very simple document. \end{document} Next, we pass the message *compile* to the object *doc*. The message *compile* is interpreted by documents of this class as a directive to run the text through the Latex formatting system [.lamport latex.]. A certain amount of output is produced by the text formatter, giving status messages and error output and the like. Once formatted, the message *print* is used to write the formatted output to an appropriate device. Note that the receive for the *print* message is the same as the receiver for the *compile* message. Although internally there may be two forms of the document, the ‘‘source’’ form and the ‘‘compiled’’ form, from the users point of view there is just one object. osh% doc compile [latex output]...

osh% doc print [printer status information]... Now let us try creating a C program. Once more, we use the message *new* passed to the object representing the class C. This time, instead of having the system prompt for the name of the new object, we give it as an argument with the command. osh% C new hello As with instances of class Latex, instances of class C can be edited (by virtue of being subclasses of class Textual). Passing the message *edit* to object *hello* places the user into a text editor, from which we can enter the classic example C program. main(){printf("hello world !\n");} Exiting the editor, we compile the program by passing it the message *compile*. As the compilation is successful (no errors were reported), we then pass the message *link* to object hello. As the *link* step was successful, we then have an executable program. Execution is obtained by passing the message *execute* to the object. osh% hello compile osh% hello link

osh% hello execute hello world !

osh% Note that some commands to the object ‘‘hello’’ may not make sense unless they have been preceded by other commands. For example, the *execute* message makes no sense unless it has been preceded by an error free *compile* and *link*. There are at least three approaches that could be taken to solve this. One would be for messages which are not appropriate to produce some type of informative error output. A second approach would be to disable such messages, such as is done on the Macintosh where currently disabled commands are displayed in gray on a menu. A third approach would be for messages such as *execute* to automatically determine which prior messages need to be performed first, and implicitly send them. Notice also that very general message selectors, such as *compile* and *edit*, are interpreted by both C programs and Latex documents, although they produce vastly different results. This is unlike the Unix world, where the user is required to remember very different commands to produce these results. Thus it is easier for the user to remember how to use the system, since he or she must only recognize the task to be performed, not how the computer should go about doing that task. And even if the programmer should forget the commands that a particular object will respond to, every object will describe itself. The programmer can use messages such as *class* and *commands* to find out more about an object. Any object that responds to the message *execute* can be made into a method; that is, the code that will be executed in response to a message. We can add a new command *hello* to class **Latex** merely by giving the class access to the executable object of the command name. Thereafter, instances of class **Latex** will respond to the message *hello*. osh% Latex addMethod hello

osh% doc hello hello world ! By using different commands new classes, new commands for existing classes, new collections (folders) can all be created within the object shell framework.

The Implementation of the Prototype Shell

The code for the prototype object oriented shell is shown in the Appendix. The program is a large, recursive bourne shell script. As might be expected, as a consequence of the interpreted nature of shell scripts, the program is not noted for its speed. Internally, each object is represented by a Unix directory. The current state of the object is maintained by files in this directory. In addition, every object contains a special file, called .class, which has as contents the name of the class to which the object belongs. All objects of a given class link together one copy of the same file, thus there is little overhead associated with maintaining this information. As we will discuss in more detail in the next section, a consequence of the fact that under Unix only files, and not directories, can be shared is that objects (which are represented as directories) cannot appear in more than one place. (In theory, symbolic links are supposed to be permitted between directories under Berkeley 4.3 Unix. We have, however, found a surprisingly large number of implementations which fail to allow this, or do it incorrectly). Thus, each object in the system can only name its class; it can not itself maintain a pointer to its class. Thus to discover the object representing the class of another object the object shell must not only print out the value of .class, but must search (using the utility *find*) for the class object so named. In response to a message, an object can do one of two things. It can either: Print a string value, usually via ‘‘Echo’’. For example, every class description will produce the class name in response to the message *class* in this manner. Perform some sequence of actions, including recursively calling the object shell. For reasons that we have just noted, an object cannot return another object; although it can return the name of another object. An analysis of how the message *edit* is handled by instances of class C will illustrate the interaction between these two types of commands. The command *edit* is not handled by the C class, but is instead inherited from the class Textual. When the object shell attempts to pass the message *edit* to an instance of class C, the test to see if the message is understood (produced by examining the directory associated with the class object, to see if there is any entry with the message name) fails. A test is then made to see if the class has a superclass, indicated by the presence of a file .superClass in the class directory. In the case of C it does, and thus we next examine the object associated with the class Textual. This time the method is found in class Textual. In object shell terms this is indicated by the presence of an object called *edit* in the data associated with the class object Textual. Since data associated with objects is maintained in Unix directories, we can use the Unix subdirectory notation to denote the object associated with this method. If we were to examine this object we could discover that it is an instance of class ShScript, a shell script. osh% Textual/edit class ShScript In Unix terms what this means is that there is a subdirectory of *Textual* called *edit*, and in the subdirectory edit there is a shell script, found in a file called *executable*. The shell script looks as follows: : edit the current object, but first find the right name vi ‘osh $* editName‘ In order to execute this code the object shell is recursively called, passing the message *editName* to the receiver of the original message. The search for a method matching this message starts once more with the class of the object, namely C. A method is found there, which looks as follows: : return the file in which the text is found echo $*/text.c Following bourne shell conventions [.bourne unix.] the file name will be captured by the command given between quote marks in the shell script for *edit*, shown above. The editor ‘‘vi’’ will then be invoked on this file. Since different classes can override the method *editName* in different fashions, in this manner the various subclasses of Textual can have independent conventions for naming their files (they must, since different processes have different conventions) but still use the same code in response to the message *edit*. For example the class Latex returns a name ending in the suffix ‘‘.tex’’ in response to the message *editName*. The class Textual itself provides a method for this message, echoing the string ‘‘text’’. Thus unless overridden by a subclass, this will be the name of the text file. Objects can be moved. For example, to become a method in a class an object is merely moved into the directory for the class. Similarly objects can be moved into and out of folders. However, as we have already noted, the fact that links cannot be made to directories means that objects cannot be shared.

Problems with the Object Shell Approach Even using the simple object oriented shell provided by our initial prototype, there are certain problems that can be identified in integrating the object oriented approach and Unix. As these problems would even apply to our envisioned graphical shell, they are worth noting. The most significant problem is caused by the necessity to impose the object oriented method on top of the existing Unix system. The notion of object is not part of the Unix basic vocabulary, and thus we must structure objects in terms of more primitive datatypes. In our case, we chose to represent an object by a directory containing a method file. While this permitted us to use the rich set of existing Unix tools in implementing the Object Shell, it prevented us from making objects into true first class entities. Since it is not possible to create or move unnamed directories under Unix, it is not possible for a method to return an object as the result of a message request. It can create a new object and return its name as a string, but that is not the same thing. Until we have devised some method for allowing objects to return other objects, progress on implementing the visual shell programming idea is effectively stymied. Finally, we note that the method bodies are interpreted in the raw Unix world, and not (unless the object shell is specifically invoked) in the Object Shell world. Thus creation of method bodies requires a mastery of both paradigms. We view this state of affairs as not entirely and necessarily bad. Unix has proved itself to be a superior software development environment; as evidenced by the prototype object shell itself, it possesses a powerful and flexible set of tools that permit new software to be easily and quickly constructed. The Unix system has not, however, been noted as an especially user friendly system, particularly to novice users. The object shell, on the other hand, provides a simple to use and easy to understand gloss over the bare Unix system, and thus permits novice users to more easily use the computer to solve their particular problems. We envision the development of a multilevel system. Casual or novice users could solve their problems entirely in the Object Shell world, never needing know the subtle intricacies of the Unix system. Application developers, on the other hand, would live with one foot in both worlds; using the raw Unix system to create new applications, which would then be incorporated into new or existing object methods. Between these two extremes would be those individuals designing object classes; they would have to know how to use Unix applications and processes, but not necessarily how to construct them.

The Next Step

There are three obvious directions for continuing in the development of a true object oriented shell, and we are progressing on all three fronts. Recoding the prototype object shell in C, instead of Bourne shell, could result in a considerable speed improvement. This improvement in speed would be due not only to the differences between a compiled and an interpreted language, but because a C program would probably implement more features (such as method lookup) directly instead of using existing tools. The graphical presentation of the contents of folders, and the selection of messages using the mouse, is largely just a practical shorthand for transmitting the information to the object shell. Nevertheless, from the user's point of view it is an important shorthand. The difficulty here is that bitmapped graphics and mice are not standardized under Unix, and thus the resulting program will be very device dependent. However, we will only be able to evaluate how well users like a graphical interface for Unix when we have an actual working system. We need to extend the range of applications that can be used under the object shell to include such commonplace Unix tools as the mail system, electronic bulletin boards, database systems, and other computer languages. Finally, there is clearly a limit to how far we can go in making a completely object oriented system if we are constrained to working on top of the existing Unix framework. This is most easily seen by considering the difficulty we have in implementing sharing or copying of objects, given our implementation technique of implementing objects by means of directories. A longer range project envisions the development of a completely new operating system, in which objects are the basic building blocks of all operations. The development of such a system would be a massive project, both in size and resources. Before undertaking such a task, there are important lessons that can and should be learned from much more limited experiments, such as the one described in this paper.

Conclusions

We have described what we think an object oriented shell for Unix should look like. As a first small step in the development of that tool, we have developed a prototype simple object shell, using the Bourne shell language. Experience with this prototype has convinced us of the usefulness of the approach, but has also pointed out some fundamental conflicts between the object oriented approach and the Unix philosophy. It is clear that a true object oriented shell can be constructed, and we are continuing work in that direction. What is less clear is how similar to Unix users will perceive the resulting system to be. Acknowledgements Paula Hannan suggested the dialog box solution to the troff dilemma. Several useful and important changes to an earlier draft of this paper were suggested by the referees.

References

Appendix

Source code for the Object Shell

```
: prototype object shell
: written by tim budd 12/86
:
export receiver
osprompt="%osh"
: examine the command
if test 0 -eq $#
then
        : no arguments, go into loop accepting commands
        ls
        echo -n ${osprompt} ""
        while read cmd
        do
                if test -n "${cmd}"
                then
                        : if there is a command, recursively do it
                        osh ${cmd}
                fi
                echo
                ls
                echo -n ${osprompt} ""
        done
        : finish off last line
        echo
else
        : grab receiver for message
        receiver=$1
        shift
        : grab message, give error if not there
        if test 0 -eq $#
        then
                echo missing message 1>&2
                exit 0
        fi
        msg=$1
        shift
        : method is taken from class of receiver
        if test ! -r "${receiver}/.class"
        then
                echo ${receiver} is not an object
                exit 0
        fi
        className=‘cat ${receiver}/.class‘
        while test -n "${className}"
        do
                class=‘find . -name ${className} -print‘
                if test -d ${class}/${msg}
                then
                        : we found it, see if it is executable
                        if test -r ${class}/${msg}/execute
                        then
                                : do it
                                ${class}/${msg}/execute ${receiver} $*
                        else
                                echo method for ${msg} is not an executable
                        fi
                        exit 0
                elif test -r ${class}/.superClass
                then
                        className=‘cat ${class}/.superClass‘
                else
                        : exit loop
                        className=
                fi
        done
        echo ${receiver} does not respond to ${msg}
        exit 0
fi
```