In *Visual Object-Oriented Programming.* **M. Burnett, A. Goldberg and T. Lewis, editors. Prentice Hall Inc., 1995**

Interface Issues in Visual Shell Programming

Francesmary Modugno
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
E-mail: fmm@cs.cmu.edu

Abstract

A visual shell, such as the Apple Macintosh Finder, is a direct manipulation interface to a file system. Although such systems are easy to use, they are are difficult to program, especially for non-programmers. This paper introduces Pursuit, a visual shell design that combines visual program specification, visual language representation and object-oriented techniques, to address some of the problems of visual shells, and discusses some of the issues relevant to making Pursuit's approach viable. First, Pursuit extends the concept of a concrete, directly manipulable object to all interface objects, so that the output of any utility can be selected as input (*i.e.*, piped) to other utilities. Second, Pursuit incorporates a Programming by Demonstration (PBD) system into the interface. This enables non-programmers to construct programs by manipulating representations of file system objects, such as files and folders, directly in the interface. Finally, Pursuit introduces an object specific, state-based visual representation language for programs. Programs are static, editable representations of objects and behaviors that reflect the interface. Hence, Pursuit provides a uniform model – that of object and behavior – for users to learn in order to understand, navigate and program in Pursuit.

1 Introduction

A visual shell is a direct manipulation interface to a file system. Examples include the Macintosh Finder and the Xerox Star. While visual shells have made computers easier to use, they have some well recognized limitations. There is no easy way to write quick "pipeline" programs directly in the interface, or to access easily the functionality of complex data manipulating utilities such as the Unix utilities *awk* and *sed*. In addition, current visual shells are difficult or impossible to program, even for expert programmers. The ability to automate tasks is almost completely lost.

This paper introduces Pursuit, a visual shell design that combines visual program specification, visual language representation and object-oriented techniques, to address some of the problems of visual shells, and discusses some of the issues relevant to making Pursuit's approach viable. First, Pursuit extends the concept of a concrete, directly manipulable object to all interface objects, so that the output of any utility can be selected as input (*i.e.*, piped) to other utilities. Second, Pursuit incorporates a Programming by Demonstration (PBD) system into the interface. This enables non-programmers to construct programs by manipulating representations of file system

objects, such as files and folders, directly in the interface. Finally, Pursuit introduces an object specific, state-based visual representation language for programs. Programs are static, editable representations of objects and behaviors that reflect the interface.

Although both the extended object model and the PBD specification technique are not original to Pursuit, this work is unique in its attempt to combine and extend these technologies in the visual shell domain. One objective of this paper is to examine the many technical issues raised by combining these technologies. While some of these issues are addressed in this or other work, several remain unsolved and require deeper investigation. Furthermore, this work emphasizes providing power to the *end user*, not the application programmer or system maintainer. Its focus is on the user interface, not on the underlying operating system. Finally, the object specific, state-based visual programming language is an innovation of Pursuit.

2 Today's Visual Shell

The concept of a "direct manipulation" interface, in which objects on the screen can be manipulated using the mouse and keyboard, was introduced in 1981 by the Xerox Star, and popularized by the Apple Macintosh. The success of direct manipulation interfaces is often attributed to the constantly visible, concrete, familiar representation of data objects. For example, a file is not just a string of text denoting a name. Instead, it is a manipulable object in the interface which resembles the real world analogue that it represents. Unlike textual interfaces, users need not guess whether what they see is a string representing a file name or just a line of text. Instead, objects in the interface can be easily identified by looking at them, just as objects in the real world can be identified by their appearance.

In addition to concrete representation, visual shells give the illusion of concrete manipulation of objects. For example, to move a file, the user drags it to its destination. Unlike textual interfaces, users do not need to remember command names and syntax. Hence, the analogy between an object's look and behavior in the physical world, and its representation and corresponding behavior in the interface, enables users to transfer some knowledge from their real world experience to their computer task.

However, while visual shells have existed for over ten years, current systems fall short of exploiting the full power of the direct manipulation paradigm. For example, only files and folders on the desktop can be manipulated because only they are viewed as "objects." All other information, such as dates and sizes, is viewed as text that cannot be manipulated in the tradition direct manipulation way.

Similarly, many utilities only output text strings in a dialog box. This text is neither selectable nor manipulable. Thus, users must often do extra processing to complete a task. For example, the Macintosh *Find File* utility (prior to System 7.0) outputs a dialog box containing a list of file names. Our experience has shown that users think of these strings as representing the actual files, not just as a list of strings. Assume that a user wish to copy those files. To do so, the user must open the appropriate folders, locate the corresponding files and then execute the *duplicate* command. Not only is this extra work, but it can also introduce errors, since the user may miss a file or open the wrong folder.

Finally, visual shells are limited in their programmability. Programming, if it exists at all, is most often achieved by using a standard textual language. Although some systems contain a macro

recorder to make a transcript of user actions that can be generalized, users must edit, and therefore understand, a representation of the program in a textual programming language to generalize the transcript. This limits programming to those with appropriate skills. In addition, these textual languages do not take advantage of the unique visual aspects of the interface. The conceptual simplification gained by the visual interface is lost when programming it. This makes visual shells more complex: users interact with the system visually, but program it in a textual language. Hence, users need to develop two very different bodies of knowledge: one for interacting with the system, and one for programming it.

3 Related Work

While commercial visual shells (*e.g.*, NeXT, HP Vue and Looking Glass) have provided a Macintosh-like desktop for Unix, more advanced features, such as pipelines, utilities and shell programming, are not accessible in the visual shell interface. Instead, they are only available through a conventional textual shell window, which requires users to already be familiar with Unix to use these features.

Some visual shells contain a macro recorder (*e.g.*, QuicKeys2, MacroMaker and Tempo II for the Macintosh, and HP NewWave) that makes a transcript of user actions which can be replayed later. Although effective in automating simple, repetitive tasks, macro recorders are limited because they record exactly what the user has done – only the object that is pointed to can be a parameter. The transcript consists of a straight-line sequence of commands. Macro recorders cannot automatically generalize, so the program cannot be applied to other objects. SmallStar [Halbert 1984] is a macro recorder that produces an "English-like" transcript of user actions containing icons for objects such as files and folders. The transcript can be generalized via a menu of commands. Users must determine where to add control constructs and how to generalize data.

Some visual shells have invented a special graphical programming language ([Henry and Hudson 1988], [Haeberli 1988], [Borg 1990]) to enable users to write programs. Most of these languages are based on the data flow model, in which icons represent utilities and lines connecting them represent data paths. Most contain no way to depict abstractions or control structures. The types of programs users can write are quite limited. In addition, these languages require users to learn a special programming language whose syntax differs significantly from what they see in the interface, and writing programs differs from the way users ordinarily interact with the system, requiring significant extra learning time.

Dynamic Windows [Mckay et al. 1989], which has its origin in the Genera operating system [Weinreb et al. 1987], is a user interface management system that introduced the idea of an extended object model. Interface objects are associated with underlying semantic objects, which are arranged in an object type lattice. Applications are defined in terms of the methods on the objects in the type lattice. The main difference between this work and Pursuit is that Pursuit's focus is on usability and understandability by the end user, whereas Dynamic Windows and Genera focus on lower level issues from the perspective of application programmers and operating system builders. Furthermore, Pursuit is investigating the use of visual language techniques for programming, whereas Dynamic Windows explored traditional textual language techniques.

4 The Pursuit Visual Shell Design

Pursuit is our attempt to address the limitations of visual shells by fully exploiting the direct manipulation paradigm. This is achieved by combining techniques (both old and new) from the visual programming and object-oriented worlds. Visual programming and object-oriented techniques try to present a clear, simple conceptualization of complex behaviors. This is exactly what is needed in the visual shell domain: a way to simply and concisely present the complex working and behaviors of the underlying file system so that the full power of the computer is more accessible to end users.

Pursuit realizes its goal using three techniques. First (section 4.1), we extend the concept of a concrete, directly manipulable object to all interface objects. In Pursuit, all interface objects, and thus all utility outputs, can be directly manipulated. Therefore, users can create straightline programs (pipelines) by directly manipulating the interface. Second (section 4.2), we incorporate a Program by Demonstration system into the interface. Programming by Demonstration is a technique for specifying abstract programs concretely. This extends the direct manipulation paradigm to include a consistent method for program specification. Finally (section 4.3), we present an editable, state-based visual language to explicitly represent program code. The visual language is designed to reflect the objects in the interface. This provides a way for users to create abstract programs.

4.1 The Pursuit Interface Model

In Pursuit, interface elements are real objects. Any object in the interface, such as a file, folder, login-id, network address or date, can be manipulated in the traditional direct manipulation way. Utilities are specified by first selecting the input objects, and then either directly manipulating them using drag and drop, or selecting a menu item. The meaning of an action or menu selection is defined by the methods of the object to which it is applied. For example, selecting a file and choosing the *sort* routine causes the file's contents to be sorted, whereas selecting all the owner objects in a folder and choosing *sort* causes the files in the folder to be reordered based on the lexigraphical ordering of their owner objects' strings.

Because Pursuit elements are treated as actual objects, all utilities output manipulable objects. The objects can be Pursuit basic objects, or they can be new objects composed of other objects or defined by the utility. The output of any utility can be selected and used directly as input to another utility. This allows users to create dynamic programs directly in the interface, much the same way that Unix pipelines are constructed.

In programming, the concept of object classes or types can be difficult for non-computer scientists to understand. This is especially true in the textual domain. For example, a variable's name often provides little indication of its type or class, which is usually defined at its declaration. On the other hand, a visual representation of an object, such as a file, often provides enough information to indicate its type. Most non-obvious objects, such as dates, sizes and owners, can often be inferred from their visual representations and/or surrounding contextual information. Like the real world, a visual shell has the advantage of providing easy recognition of objects just by looking at them. By extending the concept of an object to all interface elements in a visual shell, we extend and take advantage of the concreteness provided by the direct manipulation paradigm.

4.2 The Pursuit Specification Language

The second objective of Pursuit is to extend the direct manipulation paradigm to enable programs to be specified by manipulating data objects directly. Specifying a program is a way of defining its functionality. Traditionally, a program is specified in some static programming language. The language may be textual or visual. In this way, a program's functionality is declared completely before it is executed.

Alternatively, a program's functionality can be specified concurrently with its execution. Programming by Demonstration (PBD) [Cypher 1993] is an example of this approach. In a PBD system, users demonstrate the functionality of the desired program on a particular piece of data. From this, the system attempts to create a general procedure [Myers 1991]. This is the specification model contained in Pursuit. Because Pursuit programs are specified by executing real actions on real data, the Pursuit specification language is identical to the interface language. This is particularly appealing in the visual domain because it allows users to specify programs in the same way that they invoke operations - through direct manipulation. Thus, by adding a PBD system into the interface, we have extended the direct manipulation paradigm in a way that enables users to specify a program in a familiar way.

4.3 The Pursuit Representation Language

PBD systems have several limitations: They can infer incorrectly. Most contain no static representation of the inferred program. Feedback is often obscure or missing. And few PBD systems provide editing facilities. This makes it difficult for users to know if the system has inferred correctly, to correct any errors, and to revise or change a program. Many of these problems can be overcome by providing users with a good representation of the program to review and edit. Such a representation must be conceptually simple, and must provide a good visualization of data objects and program functionality.

Pursuit addresses the limitations of PBD systems by presenting the evolving program in an editable, state-based *visual* language *while it is being constructed.* The Pursuit visual language [Modugno and Myers 1994] is based on the comic strip metaphor [Kurlander and Feiner 1988]. Familiar icons are used to represent data objects and changes in the data objects represent operations. Two panels are used to represent an operation: The prologue shows the data objects before the operation, and the epilogue shows the data objects after the operation. The operation

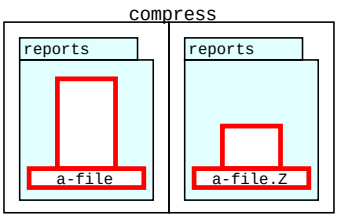

Figure 1: The representation of the *compress* operation. The file icon represents the file *a-file* located in the *reports* folder. The change in the file icon's height and in its name between the panels imply the *compress* operation. These differences are similar to the change in appearance of the real file icon users see in the actual user interface when the compress operation is executed: The file's icon is replaced with a shorter one and ".Z" is added to its name.

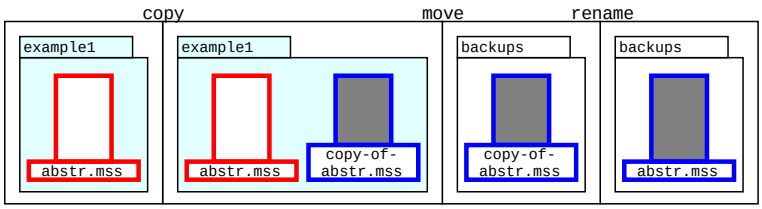

Figure 2: A Pursuit program that copies the file *abstr.mss* in the *example1* folder (panels 1 and 2), moves the copy to the *backups* folder (panels 2 and 3), and deletes the "copy-of-" prefix from its name (panels 3 and 4).

is implicitly represented by the changes in the explicit representation of the data objects (see Figure 1). An operation's representation, therefore, looks very much like the changes the user sees in the real user interface when executing the operation. A program is a sequence of operations concatenated together along with representations of control constructs and variables. Examples of Pursuit programs are shown in Figures 2 and 3.

Whereas all existing visual (and textual) programming languages explicitly represent program operations and usually require users to imagine the data in their head, the Pursuit visual language is unique in that it depicts the data objects of a program and describes operations in terms of these objects. By emphasizing the data, Pursuit's visual language taps into the object nature of the interface. Rather than thinking about a variable, its value, and how to operate on it, users relate a program object and its behavior (in terms of its visible state changes), to the associated object and its behavior in the interface. In essence, programs are symbolic representations of changes to data objects over time. They become concrete representations of behavior rather than abstract specifications of functions.

It is important to note that the Pursuit visual language is not an object-oriented language. Rather, it is a visual language that focuses on the object nature of the interface, and incorporates some object-oriented techniques within its design (further described below). Hence, it is classified as an object-based language [Wegner 1989].

5 Benefits of Pursuit's Approach

The goal of Pursuit is to provide an interface that enables users to easily access the power of the underlying system. By combining techniques from the visual programming and object-oriented worlds, we hope to achieve an interface that provides this power. We have identified three potential ways that this goal may be reached: visual pipes, utility functionality and programmability.

5.1 Visual Pipes

Programming can loosely be defined as a finite sequence of operations on data objects. We can therefore consider programming to encompass any series of user-initiated actions regardless of whether an artifact is created to later re-execute. In the Unix world, for example, writing a command sequence or pipeline can be considered a form of programming. The existence of pipes is one reason for the success of Unix among expert users [Hanson et al. 1984]. Visual shells contain a

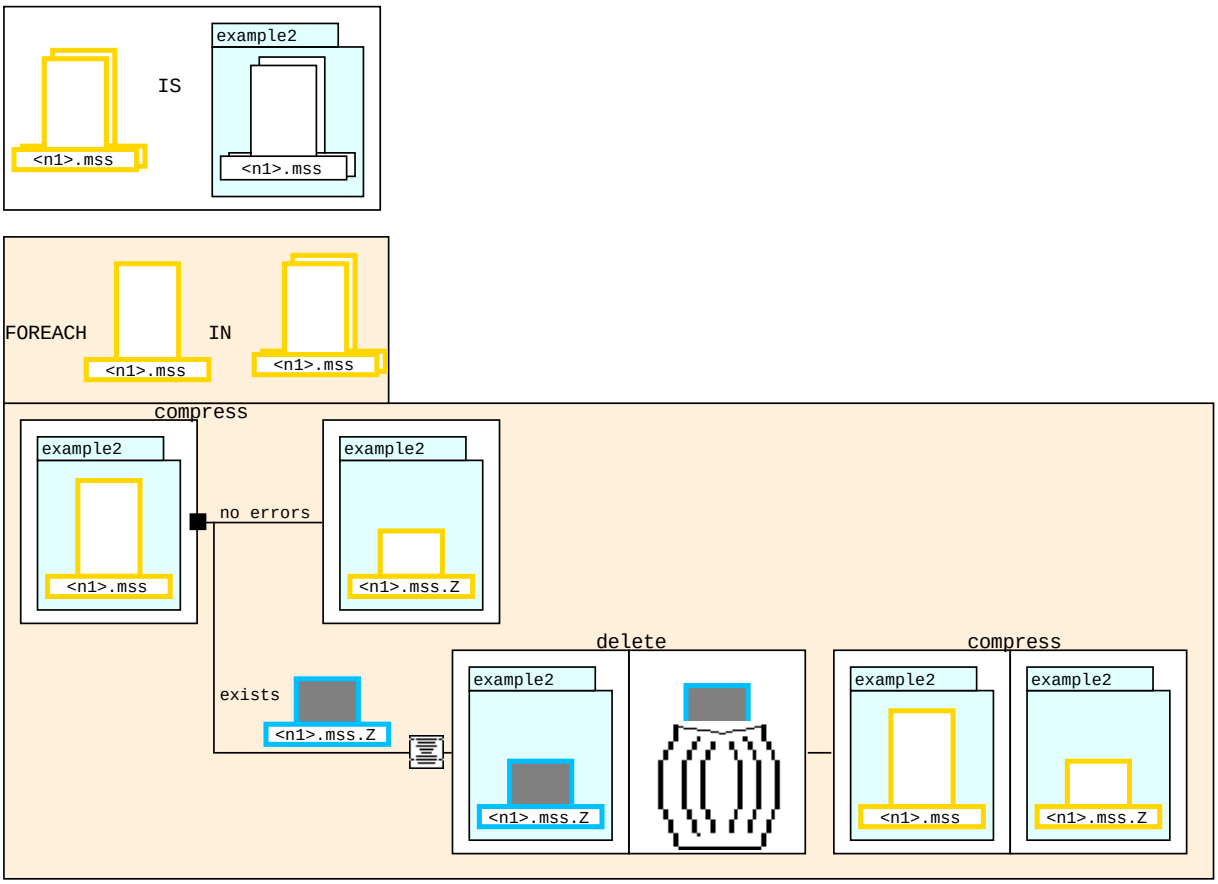

Figure 3: The Pursuit program that compresses each *.mss* file in the *example2* folder. If the compress fails because of the existence of a file with the output file name, the program deletes that old output file, and re-executes the compress operation. To produce this program, the user must demonstrate its actions on two actual file objects: one in which the compress operation executed successfully, and one in which the output file object already existed in the *example2* folder. This is depicted graphically by the visual branch and predicates after the prologue for the compress operation. The panel in the upper left corner is a declaration. It defines the set of files the loop executes over to be all the *.mss* files in the *example2* folder.

similar form of this type of programming. However, it is limited only to file and folder data objects. In addition, it is inhibited by the need for users to continually stop, examine each utility's output, and associate this output with subsequent data objects. Little support exists for the "piping" mechanism.

Pursuit expands the concept of pipes in the visual shell domain. Because all utilities output objects and not just text, users can select all or part of the output of one utility and use it as input to other utilities directly. This enables them to construct visual "pipelines" in the interface. This power arises from the Pursuit interface model, which implicitly provides a symmetry of input and output forms for all utilities. Existing visual shells cannot support complete pipelining because they lack this symmetry. Usually, programs take as input iconic representations of objects (such as files), which can be directly manipulated, but often output streams of text (such as dialog boxes), which cannot be directly manipulated. Thus, they can not use the same mechanism for pipes as for normal command invocation.

Visual pipes are illustrated in the following example. Assume a user wishes to delete all those files containing the string "typed output" from the current directory. The user begins by invoking the Pursuit search utility and giving it the string "typed output." The utility creates a window containing a collection of file and string objects, as shown in Figure 4. The files in the window are references to the actual files they represent. To remove the actual files, the user clicks on the icons in the *search* output window, and drags them to the trash – the same sequence of actions the user would use on the actual files to remove them. The user did not need to know any special languages or special flags to utilities. Instead, this power is accessed using a simple sequence of clicking, dragging and dropping actions.

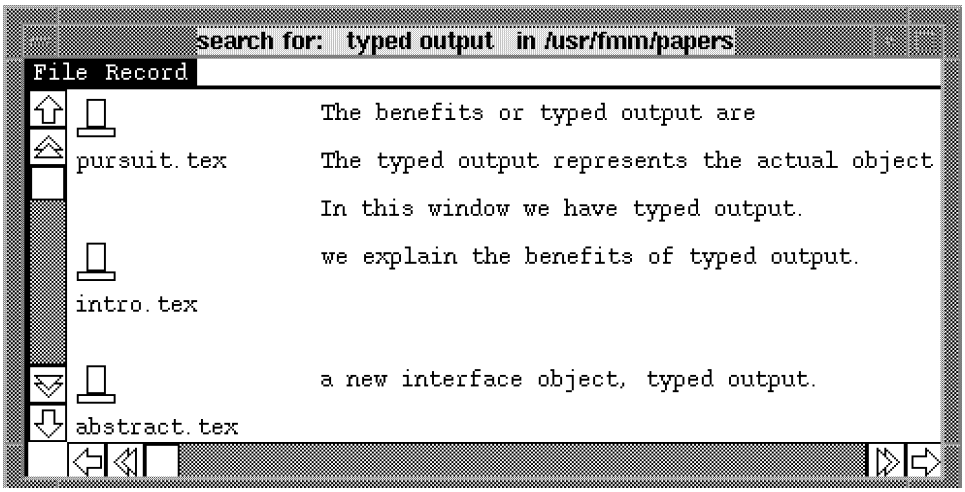

Figure 4: The output of the *search* utility containing both file and string objects. The file objects are references to the files they represent. The strings are the places in those files where the searched string was found. Users can perform operations on the referenced files and have the operations affect the actual files. They can also perform operations on the string objects. In particular, double clicking on the string object with value "*The benefits of typed output are*" opens the *Pursuit.tex* file scrolled to the position in the file containing that string.

An equivalent Unix shell command sequence is *rm 'grep "typed output" * | awk −F: '{print $1}' | sort −u'*. It is similar in form to a shell script taken from a random sampling of shell scripts written by members of the Carnegie Mellon University School of Computer Science. This particular sequence requires users to know four utility names, two flags, and one special programming

language, in addition to the correct way to format the command and how the output of each utility is structured.

5.2 Utility Functionality

One problem with most visual shells is that to achieve greater power, more and more utilities are introduced into the system. This requires users to learn new and different commands, arguments, and languages, increasing the complexity of the system.

The Pursuit interface model allows users to express some complex utility functionality in a simple way–by applying interaction techniques to interface objects. Since users are familiar with the objects, the interaction techniques, and the corresponding behaviors, this power is provided without requiring them to learn anything new. In addition, this power is provided directly, rather than by explicitly invoking utilities. For example, in the *search* example of the previous section, the user implicitly accessed the column selection functionality incorporated in the Unix utility *awk* by interacting with the output file objects directly.

5.3 Programmability

In addition to visual pipes, Pursuit handles more traditional programming. By incorporating a PBD system along with an editable, graphical representation in the interface, users can write, edit, save and re-execute programs with little programming skill. Figures 2 and 3 are examples of Pursuit programs.

The Pursuit visual language provides many useful features. First, it focuses on representing data objects. Both the class of an object and its observable behaviors are visually depicted. Each class of objects in the interface has a unique representation in the visual language. Users can identify an object's class just by looking at its icon. From an object's icon and the changes reflected in it, users can deduce an object's behavior.

In addition, all visual references to an object are instances of the prototype in the object's declaration. In Figure 3, the prototype of the set of files object with name value *<n1>.tex* is declared (in the upper left hand panel) to be the set of all file objects in the *example2* folder whose name object has value *<n1>.tex*. An instance of this set object serves as the loop set object of the *foreach* loop. Similarly, the single file object with name value *<n1>.tex* is defined as the loop iteration object. Instances of this object, along with their associated behaviors (state changes), appear in the panels for the *compress* operations. By editing an object's prototype, such as changing it name or location, all instances of the object become immediately updated.

6 Issues Raised by Pursuit's Approach

In order to determine the viability of this approach to a visual shell, we implemented a small prototype of Pursuit using the Garnet [Myers et al. 1990] toolkit. The examples discussed throughout this paper illustrate some of the important technical issues uncovered by the prototype. In this section, we discuss them and where possible suggest how a real implementation might address them. However, many of these issues require further investigation.

6.1 Providing Mechanisms For User Communication

While applications can predefine the meaning of actions on a given object, the user may wish the meaning to be something different, or may actually believe it to be something different. For example, in the search example in section 5.1, the user dragged the file references to the trash and the *original* files were deleted. However, what if the user only wants to delete those references, and leave the original files unchanged? How could the user communicate this to the system? Furthermore, assume that instead the user drags the references to another window. That could move the original files, create copies of them, or created links to them. All three are reasonable outcomes, yet a utility should do only one. Which one should be the default, and how can the user override this default behavior? These questions can best be answered through user studies. For example, we can present several scenarios to the user. Each scenario would contain a different default behavior for an action on utility output. We can then ask the user which action seems "right." The most common "right" behavior could be the default. We could then investigate other ways to allow the user to specify different behaviors, such as through the use of modes.

6.2 Automatically Coercing Types

Sometimes a user may want to use an object as input to a utility, although the object may not match the utility's input type. For example, the user may wish to use an object, other than an e-mail address, as the input to the mail utility. The value of this object is a valid mail address, such as as a file owner object. How can we do this type conversion automatically? The object-oriented solution is to send the object a message requesting its value in the required type. Only objects that respond to the value request message could be valid arguments to a utility, and as new object types are introduced into the system, new methods must be added to existing objects so that they can respond to requests for values of the new type.

Another approach, found in Dynamic Windows [Mckay et al. 1989], is to require all coercions to be defined explicitly by the application programmer. That is, each utility must provide a *translator* from any object type to the utility's input type. A limitation of this approach is the growing number of translators needed as new objects and utilities are introduced. It also makes extending the system more complex.

Alternatively, the shell can do this "translation" automatically by requiring that all objects have a method that returns its value as a string. When a conversion must be made from one type to another, the shell can get the string value of the first object and then locate in the system the object of the second type having the returned string value. Unfortunately, there is no guarantee that this approach will return an object or that it will return only one object. Nor is it certain if it is general enough to handle all coercions from one type to another.

Another issue that must be explored is how the user will perceive this automatic type coercion. Do users need to be aware of the type conversion or should it be invisible to them? Is the "magic" of automatic type conversion confusing to the non-programming user?

Finally, further research must be done to determine how (if at all) type coercion should be represented in the visual language.

6.3 Providing An Extensible Application Language

One of the most difficult yet very necessary properties of an interface is extensibility. In addition to allowing for the introduction of new utilities, we must make provisions for the introduction of new object classes and new interaction techniques, both of which must work successfully with the existing system. There are many issues that must be addressed in order for the system to be extensible. For example, suppose a new interaction technique is introduced into the system. There must be a way to define the meaning of this action on existing system objects. Similarly, if a new object is introduced into the system, care must be taken to insure that the meaning of all interaction techniques on this object is defined.

The Pursuit visual language must also be extensible to represent new utilities. To address this problem, we have developed a declarative language for specifying the visual representation of utilities that operate over file and folder objects. An interesting question is how to extend this language to cover polymorphic utilities and utilities that operate over objects other than files or folders.

6.4 Making the System Scalable

Like all visual languages, Pursuit's representations can grow large rather quickly (for example, see Figure 3). In order to allow users to write more complex programs, some techniques for making programs compact must be employed. One approach, found in Chimera [Kurlander and Feiner 1988], is to incorporate more than one operation into a single panel. In Chimera, all operations specify how they can be visually combined with other operations. Unfortunately, this approach is limited because it is not easily extensible. Whenever a new application is added to the system, all other applications must be updated to indicate how they can be visually combined with the new operation.

Pursuit takes an alternative approach. Rather than having each utility determine when it can be combined with other utilities, all Pursuit utilities must declare which data objects they affect and how they alter them. Pursuit uses heuristics that combine knowledge of the domain with this declarative knowledge about operations to make programs more concise. For example, Pursuit determines when it can combine the prologue of an operation with the epilogue of the previous operation. In Figure 2, the second panel serves as both the epilogue for the *copy* operation and the prologue for the *move* operation. Pursuit also uses these heuristics to determine when several operations can be represented in a single panel. The shadow beneath the third panel of Figure 5 indicates that it contains both the *move* and *rename* operations. By clicking on it, users can see the individual panels for the two operations, resulting in the program of Figure 2.

Another way to address this problem is to allow users to select a set of panels to collapse into a single panel. The problem here, however, is how to visually represent a user-defined composite panel in a way that is consistent with the visual language. Unlike panels collapsed automatically, the user may select a set of uncompatable panels to collapse together. For example, the panels may contain two operations that change the same property of an object, such as a file name. It would be difficult to determine which of these changes the composite panel would reflect. Furthermore, if the set of operations contained within the panels manipulate multiple, unrelated objects, it would be difficult to represent clearly the different operations and the objects they affect.

Another space-saving technique that can be incorporated into the visual language is to auto-

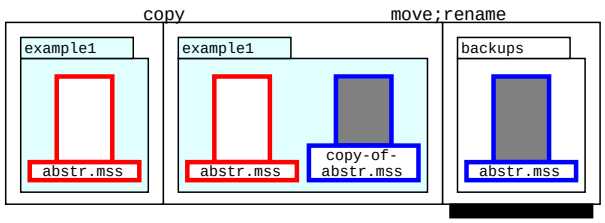

Figure 5: A more concise representation of the program in Figure 2. The third panel contains multiple operations, as indicated by its shadow. Clicking on the shadow reveals the panels for the individual operation, producing the representation in Figure 2.

matically collapse loops into a single miniature panel that could be expanded to allow users to examine its contents. While this approach is promising, it raises questions about how to handle nested loops and conditionals as well as questions about program layout.

6.5 Representing Conceptually Abstract Objects

Although the Pursuit visual language has an easily identifiable representation of files and folders, no such representation exists for other interface objects. Currently, all other objects are represented as rectangles containing their name and class (see Figure 6). However, these representations bear

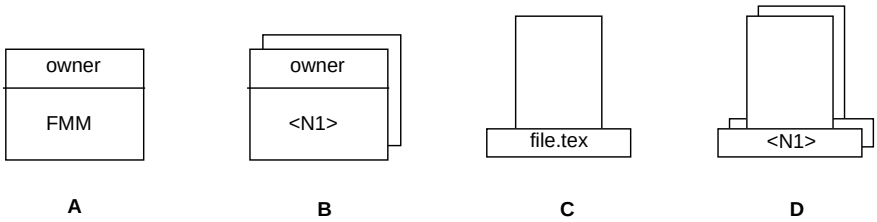

Figure 6: Examples of how different objects are represented in the Pursuit visual language: A) A single owner object whose value is "FMM". B) A set of owner objects whose value is the abstract object <N1>. C) A single file object whose name is *file.tex*. D) A set of files whose name object is *<N1>*. Notice that while the visual objects used to represent files closely resembles their representation in the interface, other interface objects do not have as evocative a representation. The "best" representation of such objects remains an open question.

little resemblance to their interface counterparts. Further research is needed to determine if there is a good graphical representation for objects such as owners, dates and sizes, in both the interface and the visual language. Furthermore, when choosing representations, care must be taken to maintain the goal of having the visual representation of programs mirror the changes in objects in the interface.

A related question is how to represent user-created objects (e.g., programs). Since users may make programs into icons, which can then be manipulated, these icons can be used in future programs (like subroutines or functions). How can we represent them in the visual language? One way is to have an abstract program object in the visual language. When one program is incorporated into another, the visual representation of the latter program can contain an abstract program object labeled with the former program's name. However, this approach strays from the goal of mirroring the interface object's representation. An alternative is to have users create their own icons for

user-defined programs, and use these icons in programs. This is similar to the approach used by Forms/3 [Burnett and Ambler 1994] for allowing users to define the appearance of user-defined types. However, users may find it difficult or even annoying to have to design meaningful icons for every demonstrated program. Alternatively, the system can automatically try to generate an iconic representation using the changes in representations of a program's objects. This can be especially difficult for programs that manipulate a large number of unrelated objects.

7 Summary

This paper introduced a new approach to a visual shell design. The approach incorporates some techniques of visual specification, visual representation and object-oriented programming into the interface to provide users with much of the desired but absent power of visual shells.

In graphical domains users often view interface elements as more than just shapes or strings of text. By making all interface elements "real," they become objects that users can manipulate. This extends to users the ability to write programs and access some common functionality of utilities directly in the interface without requiring that they know any special commands or languages.

The Pursuit PBD (Program by Demonstration) system provides a consistent form of specification and representation of programs in the visual shell domain. This gives users the ability to write abstract programs with minimal programming skills. Because the visual language explicitly represents data and implicitly represents operations as changes to data objects, the visual language reflects the objects and changes users see in the interface when operations are executed.

When manipulating objects in the interface, operations are centered around objects whose structure and behavior reflect the underlying structure and behavior of the system. Similarly, when constructing and viewing a program, computation is centered around objects whose structure and behavior reflect the structure and behavior of the interface. This provides a more uniform model – that of objects and behavior – for users to learn in order to understand, navigate and program in Pursuit.

While the extended object model and Programming by Demonstration techniques presented here are not original to Pursuit, the way in which they are combined and extended, as well as the visual language, are. Furthermore, Pursuit's approach has raised several technical issues that lie in both the object-oriented and visual language domains, such as how to provide mechanisms for determining user intent; how to convert objects from one type to another; how to make the system extensible and scalable; and how to integrate the representation of abstract objects into the visual language. While we have discussed some existing approaches to solving these problems and suggested some new ones, many require further investigation.

Acknowledgments

The author thanks Brad A. Myers for his continued support of this work. This research is funded by NSF grant IRI-98020089, and by grants from the Hertz Foundation and AAUW.

References

[Borg 1990] K. Borg, "IShell: A Visual UNIX Shell," *Proceedings of CHI '90*, Seattle, Washington, Apr. 1990, pp. 201–207.

[Burnett and Ambler 1994] M. Burnett and A. Ambler, "Interactive Visual Data Abstraction in a Declarative Visual Language," *Journal of Visual Languages and Computing*, Vol. 5, No. 1, Mar. 1994.

[Cypher 1993] A. Cypher, *Watch What I Do: Programming by Demonstration*, The MIT Press, Cambridge, MA, 1993.

[Haeberli 1988] P.E. Haeberli, "ConMan: A Visual Programming Language for Interactive Graphics," *Computer Graphics*, Vol.22, No. 4, Aug. 1988, pp. 103–111.

[Halbert 1984] D. Halbert, *Programming by Example*, PhD thesis, Computer Science Division, University of California, Berkeley, CA, 1984.

[Hanson et al. 1984] S. Hanson, R. Kraut, and J. Farber, "Interface Design and Multivariate Analysis of UNIX Command Use," *ACM Transactions on Office Information Systems*, Vol. 2, No. 1, Mar. 1984, pp. 42–57.

[Henry and Hudson 1988] T. Henry and S. Hudson, "Squish: A Graphical Shell for UNIX," *Proceedings of Graphics Interface '88*, pp. 43–49.

[Kurlander and Feiner 1988] D. Kurlander and S. Feiner, "Editable Graphical Histories", *Proceedings of the IEEE Workshop on Visual Languages*, Pittsburgh, PA, Oct. 1988, pp. 127–134.

[Mckay et al. 1989] S. McKay, W. York, and M. McMahon, "A Presentation Manger Based on Application Semantics". *Proceedings of the ACM SIGGRAPH Symposium on User Inteface Software and Technology*, Williamsburg, Va, Nov 1989, pp. 141–148.

[Modugno and Myers 1994] F. Modugno and B. Myers, "Visual Programming in a Visual Domain," Carnegie Mellon University Technical Report CMU-CS-94-109, Jan. 1994.

[Myers et al. 1990] B. Myers, D. Giuse, R. Dannenberg, B. Vander Zanden, D. Kosbie, E. Pervin, A. Mickish, and P. Marchal, "Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces," *IEEE Computer*, Vol. 23. No. 11, Nov. 1990, pp. 71–85.

[Myers 1991] B. Myers, "Demonstrational Interfaces: A Step Beyond Direct Manipulation," *IEEE Computer*, Vol. 25, No. 8, Aug. 1992, pp. 61–73.

[Wegner 1989] P. Wegner, "Learning the Language," *Byte Magazine*, Vol. 14, No. 3, Mar. 1989, pp. 245–250.

[Weinreb et al. 1987] D. Weinreb, J. Walker, D. Moon and M. McMahon, "The Symbolics Genera Programming Environment," *IEEE Software*, Vol. 4, No. 6, Nov. 1987, pp. 36–45.