UNIVERSITY OF SOUTHAMPTON

On the Visual Programmability of Desktop Interfaces

Donald Gavin Cruickshank

Doctor of Philosophy

Department of Electronics and Computer Science

October 2003

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING AND APPLIED SCIENCES

ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

ON THE VISUAL PROGRAMMABILITY OF DESKTOP INTERFACES

by Donald Gavin Cruickshank

Today's modern desktop interfaces allow us to utilise a computer by the direct manipulation of iconic representations of the computer's resources. As computer resources and our requirements increase, we find the need to create scripts for common tasks. In this thesis, we refine the Prograph Shell. *Psh* allows us to create visual scripts in the same language as the existing direct manipulation of the desktop metaphor. In the process of creating object oriented variants of Psh, we discover that direct access to dispatching algorithms allow us to implement novel features into the class structure of the Psh environment. We look at the gestures commonly found within desktop interfaces, and construct a new gesture that facilitates the creation of UNIX style pipelines.

Table of Contents

List of Figures

1. Introduction

The current desktop metaphor has only a basic subset of the functionality of command line interfaces, such as the UNIX shell. It offers a convenient interface for the casual user, but is usually limited to simple job control. The desktop contains iconic objects that resemble items found in the office, such as documents, folders, or the trashcan, and they are presented to the user in a direct manipulation environment. This environment depicts the current state of the computer, but it displays few cues about what has happened or what may happen.

Direct manipulation of graphical icons is intuitive; an icon can have a pictorial representation that describes its behaviour. For example, dragging a document into the trashcan implies that the document will be lost when the trashcan is emptied. Such systems are easy to learn, especially when compared to the environment offered by a typical UNIX shell. The problem with the current desktop metaphor is that the direct manipulation becomes labour intensive when the user has more demanding requirements. In particular, as typical disks have grown from less than a gigabyte to hundreds of gigabytes and networked computers are much more common, the user requires the larger-scale tools from larger-scale interfaces, such as UNIX or VMS.

If the user wants to print all documents in a folder with names starting with the word "report" using direct manipulation techniques, then the user has to look at the folder, manually select the required files, and then drag them to the printer. In contrast, UNIX users might enter the following:

```
lpr report*
```

In this case, the shell finds the files that start with "report", and copies them to the printer. The problem is that the power of the desktop metaphor does not scale as well as the UNIX environment. AppleScript [AppleScript 94] adds scripting ability to the Finder, but requires the user to change to a text based programming language. An alternative to learning textual scripting languages is to acquire a large range of utilities. A larger range of utilities would increase the actions that you could perform in the desktop environment, but the user may still not get exactly what they want.

In the Finder, actions can be sequenced and stored as a script by *recording* from the AppleScript editor. Most systems have an equivalent facility; in a Windows environment the user must use a DOS shell. However, control flow such as iteration and conditionals have to be programmed using a text based AppleScript editor. This breaks the desktop metaphor, and it seems the user is forced to program the system from "behind the scenes" within a textual language. We discuss an early prototype of our own system to allow the user to program the desktop within the desktop metaphor in Chapter 4.

The typical use of the desktop metaphor is for job control. For example, an application is started by choosing the application's icon, or by opening a document that was created with the application. The documents are just entities that are simply passed onto the relevant application. To improve the facilities, the current range of desktop objects must be enhanced to cater for general purpose processing.

One of the main drawbacks of the desktop metaphor is that there are only a small number of gestures available, such as the mouse click, double click, drag, etc. Due to the lack of gestures, some operations are accomplished with the same gesture. In the Finder, moving a folder to the trashcan and ejecting a disk share the same gesture. In contrast, the UNIX interface has many gestures, which are denoted by special characters, such as the backtick for command substitution and "$" for variable substitution.

As an example of the lack of gestures in the current desktop environment, consider a compressed document. The icon of the compressed document indicates that double clicking it will decompress the document. In the Finder, the decompression utility has a choice between deleting the original (compressed) document or not. In either case, the only gesture that makes sense in the current Finder environment to invoke the decompression process is the *Open Document* gesture. Both cases are plausible, and decompression utilities on the Macintosh typically have a flag in their preference file to select either behaviour. In addition to decompressing the document, the desktop environment may choose to allow the user to navigate the contents of the archive by opening the archive as a virtual folder.

The relevance of this research is increasing, as the normal way to access a modern computer is via the desktop environment. Many computer users have never used a command line interface, and of those who have, many will have never written a script. The novelty of the desktop has long since worn off, and Direct Manipulation of the desktop is now the typical method of managing the files on a computer.

Whereas the personal computer may be seen as straightforward, the internet is comprised of many protocols and is confusing to most people who try to comprehend the complexity of all of these networked machines. The traditional tools used, such as telnet, ftp, gopher, electronic mail clients and such serve to accentuate the complexity of the internet, and the need to use the correct tool for a specific protocol. The URL based web browser, on the other hand, ties all these protocols into a single reference scheme that a navigation tool, such as a web browser, can delegate to the relevant utilities. Internet connectivity is spreading out world wide, and the tools developed for this are also used in localised internet style networks - intranets. We discuss how we deal with the issues of protocol and content type in Chapter 5.

Computers in general are becoming a commodity item for the non computer literate person. Psh is designed to be similar to the current desktop environment in looks and to be able to downscale the interface even to the level of a simple drag and drop file manager. This is important, because the grammar of textual programming is out of the reach of most non-programmers. A dataflow graph is similar. If the user is confident with the simple desktop environment, then writing simple scripts should not be too far a step for the non-computer literate person to take.

This research differs from other component technology research in that we started from an interface (the desktop environment) that is used by non-programmers and have extended it. It is not the aim of this research to apply component technology in the large. The complexity of such a shell environment is limited by the non-programming user's ability to categorise their workspace, not a trained programmer in component technology. In Chapter 6, we discuss the technique where we can add extra functionality to Prograph and consequently Psh, to enable command objects and data objects to offer their own implementations for desktop commands.

In Chapter 7, we discuss the common gestures used in modern desktop environments and introduce our own, Drop-and-Catch, which is much closer to the UNIX philosophy of constructing pipelines. A demonstrator is built and a usability trial is performed with 10 participants. The extensive commentary from those users is presented with future direction for the interface technique.

1.1 Psh

In this thesis, we discuss ongoing development of the Prograph SHell project, "Psh". The primary objective of Psh was to augment the desktop metaphor with scripting ability, whilst remaining in the visual domain of the desktop. The early development of Psh [Glaser, Smedley 1995] focused on the similarities and differences between the Macintosh Finder and the visual programming language of Prograph. A Finder window contains a set of icons, representing files and folders, with almost no visual programmability. A Prograph method window contains a bag of icons, with dataflow arcs, that represent a visual program. Psh is an attempt to merge these two concepts by using Prograph as a mock desktop environment.

The original Psh was an attempt to enhance the desktop metaphor with visual scripting. A Finder window contains icons that are passive objects. The default gesture, *double clicking*, opens the selected documents with their respective applications. The Psh prototype allowed the inclusion of user programmable desktop machines, or commands, to be represented as icons that reside in such a window. An extra gesture, *execute*, would execute the commands in a window. Classic Psh provided the basic model of visual scripting that has been carried over to FCD Psh.

The development of Psh is discussed in three chapters of this thesis. An early version of Psh is covered in Chapter 4. The initial object-oriented version is covered in Chapter 5 and the revised object-oriented version is covered in Chapter 6.

1.2 Prograph

The implementation vehicle of our prototypes is the visual object oriented language of Prograph. Prograph is an entirely visual programming language that contains a fully-fledged interpreter and debugging environment in addition to the ability to compile visual applications to self-contained executables. In this thesis, we focus on the interpreter environment of Prograph as our prototype of the Psh environment.

The key to programming in Prograph is dataflow programming. A Prograph program consists of dataflow graphs containing nodes as operations that are connected by dataflow arcs.

Much of the Prograph environment is reflective. The class system in Prograph supports full introspection of the class hierarchies and their methods. The Prograph language is discussed in detail in Chapter 3.

1.3 DDD Psh

The DDD Psh, the data directed dispatching version, is our initial object oriented Psh prototype that utilises the Prograph OO model. As Prograph is an untyped language, types are bound to data and not to nodes in the visual language. Thus dispatching a method on a Prograph object has to be determined during each method request, hence the name *data directed dispatching*. This version of Psh is discussed in Chapter 5.

1.4 FCD Psh

The version of Psh which contains the First Class Dispatcher, FCD Psh, is a refinement of DDD Psh. In addition to classifying data, we also implement commands as a class hierarchy. We relate to our experiences in defining self contained blocks of functionality within DDD Psh, and we discuss an alternative OO mechanism to perform the resolution of effective methods. FCD Psh is discussed in Chapter 6.

1.5 Internet

Another aspect that we will consider is that internet services are becoming part of the modern desktop. With a suitable tool, such as Anarchie [Stairways 99], FTP access via the URL [Berners-Lee et al. 94] has already been integrated into the language of the Finder. Recently, Microsoft Windows has also moved towards removing the distinction between URLs and local files. This is important, because a considerable amount of time is spent using the internet/intranet to communicate, and we will consider a system in which the only file identifier is the URL.

This problem is part of a greater confusion that arises in accessing data using different transfer protocols, since there is not a clear separation between type and access method. For example, accessing a file of characters that are structured as HTML will be displayed as text or rendered differently according to the server, the transport protocol and the browser, with each making almost independent decisions.

1.6 Objectives

Our objectives in this thesis are to further investigate the technology behind Psh with regard to the levels at which commands are used. For example, the command to create a list of filenames contained in a folder would be a standard command in a shell. Slight variations, or commands that have been customised by the user need to be classed differently. In Psh, if we wanted to invoke a command in a different way, we needed to either create a new version of the command in our global toolkit, or explicitly customise the command upon each invocation. We need to find an alternate method that does not explode the size of our toolbox, nor force us to customise our commands on each invocation.

The diversity of datatypes needs to be considered. We would like to be able to use the same commands on different types of data. For example, counting the words in a text file would operate in the way we would expect. If we were to give it a picture instead, then we might want it to interpret the picture and look for words. It is envisaged that such functionality will be provided by *packages*. A new package might contain new datatypes for the Psh environment, along with methods which work on those datatypes. These packages will need to interoperate and we must find a mechanism for allowing packages to be extendable with further packages.

With integration of the desktop with internet and intranet applications, we aim to open up the desktop to network operations. At this point, we should make it clear that the Psh project was not intended to provide new internet/intranet services, but to allow the user to make good use of services already available.

The main text of this thesis is split into the following chapters. Chapter 2 discusses some of the background technology relating to the work done on Psh. Chapter 3 gives an introduction to the Prograph language, and describes the visual syntax and annotations required to interpret the graphs given throughout this thesis. Chapter 4 describes the state of Psh before the object oriented modifications took place. Chapter 5 describes the first iteration of the object-oriented version of Psh, named DDD (Data Directed Dispatching) Psh. In Chapter 6, we discuss the shortcomings of DDD Psh, and refine the model of the Psh environment to produce FCD (First Class Dispatcher) Psh. Chapter 7 discusses the interaction between the user and the Psh environment. The remaining chapters discuss related work, areas of ongoing research and the final conclusions.

2. Background

2.1 Introduction

In this chapter, we introduce some of the background of visual scripting languages for desktop environments. Initially, we look at textual scripting languages that are available for current operating systems, and an equivalent work in progress that repeats much of the early (non object-oriented) progression of the original Psh prototype, except in the Scheme language [Sussman, Steele, Gabriel 93].

We briefly discuss the network services provided by the internet protocol (IP) and the benefits of the URL as a means of unifying the access to those services. We discuss the role of the internet browser and the interaction between the browser and related services from the operating system.

Visual programming languages are then introduced, and we compare the two leading visual programming paradigms. Finally, we identify key issues of modern desktop interfaces and relate these to the concepts found in visual programming languages.

2.2 Command line shells

In this section, we look at a commonly used scripting system in the textual domain, the UNIX shell. The UNIX shell is an interactive command environment for UNIX systems. A user invokes commands to the UNIX system by a command line interface within a text-based console. Feedback from those commands is given back via the console, and the end result is that the console contains a chronological sequence of interactions that we have made with the UNIX system. This mode of operation is distinctively different from today's desktop interfaces, as the console shows us the state of the system at the time each command was processed. If we list the contents of a particular directory, it is displayed in the console, and it becomes part of the console's history buffer. If we invoke subsequent commands that will change the contents of the directory, the directory listing is out of date.

The UNIX shell has evolved over time, and includes a number of powerful features. One such feature, the pipeline, allows us quickly to create powerful commands by combining more simple commands together. Commands that can be *piped* are known as filters. They are special in that they take their input from a particular input stream *stdin* (standard input stream) and give their output to a particular output stream *stdout* (standard output stream). As the shell is aware of the standard I/O streams, it is able to create a flow of data between the output of one command and the input of another. As an example, consider that we want to sort a text file, using the third column as the sort key and we want to view the results page by page via the console. We might use the following pipeline:

```
cat datafile | sort +2 | more
```

The cat command reads the file *datafile*, and sends it to *sort*. The +2 argument instructs sort to use the third column as the sort key. The output from the sort command is given to *more*, which is a UNIX utility that allows us to interactively view data on the console page by page.

If we were to perform this sequence by typing the commands individually into the console, we would have to store the intermediate results into explicitly named files, and deal with the removal of the intermediate files when they are no longer required. The benefits of the pipeline approach is that it allows us to pipeline the processes simultaneously. If the behaviour of the filters allow for it, we can expect output from the last filter in the pipeline before all the data has been read by the first filter. The pipe descriptors have a limit to the maximum amount of data held in the pipe at any one time, thus reducing the overall memory consumption. For example the intermediate data file might have a size greater than the storage capacity of the computer. In the case of using a pipeline, the temporary file is produced and consumed simultaneously. When the pipe reaches maximum capacity, the input to the pipe is marked as blocked, and will become unblocked when the next filter reads data from the pipe. Without the aid of a pipeline, the task would fail as there would not be enough room on the filesystem to hold the temporary data. To process a sequence of tasks in the shell without the aid of a pipeline, we might use the following commands:

```
sort +2 < datafile > /t/sortedfile
more /t/sortedfile
rm /t/sortedfile
```

High level process control can be achieved by using the || (*or*) and && (*and*) operators. The example below shows a command that includes two conditionals. In this case, the || operator means that *foo* is executed first. *bar* is attempted if and only if *foo* fails. Finally, *baz* is attempted only if either *foo* or *bar* succeeded. In these invocations, a failure condition is determined by *return codes*. A non-zero return code typically indicates a failure.

```
(foo || bar) && baz
```

In addition to command invocation, the UNIX shell has the ability to execute scripts. A simple shell script would be a text file containing a special header, so that the shell identifies the file as a script, and a sequence of commands for the shell to execute. In the example above, *foo*, *bar* and *baz* could be replaced by script names, or complete pipelines. From the outset, the UNIX shell has been a complex, and often unforgiving, environment to work in. The original shell, the Bourne Shell, has programming language style constructs to support conditionals and iterations.

The *alias* feature can be used to modify the behaviour of a command by macro substitution. The example below shows the creation of a new 'virtual' command, called *print*, that prints using *two-up*. We have not created a new binary anywhere, we have simply told the shell that our new *print* command is similar to *enscript*, but with a different set of parameters.

```
alias print (enscript -2r)
```

The UNIX shell has several methods for storing complex commands for later use. Typing in a command in the form of a pipeline will be executed once. If it is required later on, it can be retrieved from the command history. This is done either by using the *history* command in shells like *csh*, or by using cursor keys in more modern shells. If we wanted to save the command for further use, we would use the clipboard to copy and paste it to a more permanent medium. A more permanent way of using a command several times within a session is to use the *alias* feature. Defining an alias in the shell persists only for the current session, as it is stored within the runtime environment of the shell. If we want to have the alias available in successive sessions, we would insert the alias definition into a script which is called during the shell's initialisation process.

The typical UNIX shell is a powerful environment for process management, but string manipulations are often relegated to utilities, such as *sed* or *awk*. Although powerful in their own right, this forces the user to switch between different languages, each with their own special processing abilities and peculiarities. The Scheme Shell [Shivers 94] is an attempt to remedy this. Shivers noted that the UNIX shell is a command environment accompanied by a mixture of specialised languages, and attempted to build a shell with a single programming language. The implementation involved taking an implementation of Scheme, and tacking on process management commands and an interface to the UNIX system functions. This work is of special interest to us, as both the Scheme Shell and Psh are built on an interpreted language environment, and then specialised towards the UNIX shell.

2.3 Internet technologies

2.3.1 Transfer Protocols

The Internet provides a wide range of services, such as electronic mail, file transfers and the World Wide Web. These services are supported by a number of Transfer Protocols, which describe the method of communication between computers on the network to perform each service. Typically, each service is associated with a port number, on which a server will listen for connections requested by respective clients. For example, the World Wide Web is served via the HTTP Transfer Protocol, which is normally associated with port 80. Most user-level internet clients will connect to a server via a mechanism supplied by the operating system's network software known as a socket. Upon successful connection between the client and the server, the communication is performed as a byte stream between the two sockets. It is at this level that the Transfer Protocols are applied.

As the data is transferred as a simple byte stream, it is un-typed. MIME (Multipurpose Internet Mail Extensions) is a scheme that was originally intended to classify components of electronic mail. Normally, electronic mail documents are assumed to be ASCII text. With multi-part mail messages, the MIME types are used to inform the mail client what type of data each part is, so that the client can perform suitable actions, such as to display pictures. The MIME classifications have been adopted for USENET news and the World Wide Web.

2.3.2 Reference schemes

Along with the World Wide Web, the introduction of the web browser also introduced the URL. The URL (Uniform Resource Locator) encapsulates the protocol we want to use, the machine that we want to communicate with and other location information into a single textual string. From an end-user's point of view, the URL greatly simplifies the access to resources on the internet, because the user needs to learn only a single reference scheme. Modern desktop environments, such as Microsoft Windows and the Macintosh Finder, have special support for URLs, and will typically have 'URL dispatchers'. This is an important step in the evolution of the internet-empowered desktop, as the end-user is able to invoke the majority of internet services from a single navigation tool. In effect, this removes the need to launch protocol specific client software, and with it, the disjoint nature of the different Transfer Protocols. In turn, the web browser either produces the relevant interface for the protocol, or delegates the task to an external application.

The URL has unified the way that we refer to resources on remote machines. However, it also unifies local and remote access by allowing local file access with the 'file:' access scheme. Current desktop environments have not embraced utilisation of the URL local file access scheme to the same extent as the other URL schemes. The reason for this, perhaps, is due to the nature of the current navigation tools. We currently regard the web browser as a tool that can be heavily influenced by external forces (i.e. servers with ill intent) in a negative way. With the introduction of Java and JavaScript, the act of viewing a web page can invoke processes that could exploit security holes that our navigation tools may contain. These security issues are an effect of those tools; not of the URL itself.

2.3.3 Navigation tools

There are numerous navigation tools that we might use on the internet. In the early days of the internet, we had to use the correct navigation tool for a given protocol, such as an FTP client for the FTP protocol, a newsreader for the NNTP protocol, etc. The modern navigation tool is the desktop environment itself, or an internet browser such as Microsoft Internet Explorer or Netscape Navigator.

2.4 Visual programming

In this thesis, we refer to visual programming as programming with visual graphs (arcs and nodes), rather than visual interface building as performed with Visual Basic. An example is the Prograph language which is given in Chapter 3, and this is what we mean to be visual programming. A Prograph programmer creates a program by drawing a dataflow graph. To give the scope of visual programming, we give a brief account of two metaphors that have been utilised in visual shells: the *dataflow* metaphor and the *comic strip* metaphor.

Figure 1 shows the basic appearance of the two metaphors. The dataflow metaphor consists of a number of elements connected via lines. Typically there will be dataflow in one direction, for example downwards. Data flows out of the bottom edge of boxes into the top edge of other boxes. The comic strip metaphor (shown on the right) does not explicitly show data transfer, but shows the order in which the processes occur.

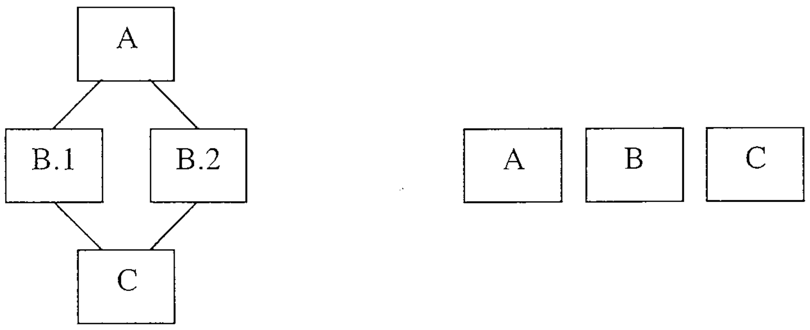


Figure 1 - The dataflow metaphor (left) and the comic strip metaphor (right)

The dataflow metaphor is based on a directed acyclic graph. The nodes of the graph describe changes that happen to the data on its journey through the graph, and the arcs show the paths of dataflow in the graph. The execution order is relaxed; the main criteria is that a node can only be executed when data is available on all of its input terminals. This is the metaphor used in Prograph [Giles, Cox, Pietrzykowski 96], IShell [Borg 90], WebFlow [Fox, Furmanski 96] and Vista [de Mey, Nierstrasz 93]. The nodes in a dataflow graph can have multiple input and outputs, and the user is able to use output from a single node as input to several other nodes. To support conditionals, a dataflow function can consist of a set of graphs, each depicting a possible outcome of the function as a whole. The dataflow model used in Prograph has control flow semantics similar to Prolog, and has the separate cases shown in different "panes". We will demonstrate this in Chapter 3.

With the dataflow metaphor, the semantics of a gesture are usually clear. For example, initiating decompression (mentioned in Chapter 1) is clear. The decompression utility receives the *contents* of the compressed document via a dataflow arc, and thus is not directly associated with the actual content of the original document. Deleting the original document does not make sense, as we have only examined its contents. If we are to store permanently the decompressed copy, we then move it back to the file space.

The comic strip metaphor is based on a comic strip. It consists of a sequence of pictures that represent a sequence of changes to the data. Although it is simple in concept, it is a convenient way to show output from a simple macro recording. This feature has been utilised in Pursuit [Modugno et al. 95] and Silk/JDI [Landay 96]. To denote control flow, Pursuit has used annotated arcs between different *comic strips*. In this form, the comic strip metaphor is equivalent to a flowchart.

It is also worth noting that each process in the comic strip metaphor processes only a single input and output. If we need to split the data, process only part of it and rejoin the results to produce a single result later in the strip, we need to temporarily store the portion of data that is not to be processed elsewhere.

2.5 Desktop environments

The desktop environment is the normal method to interact with modern desktop computers. Desktop environments offer virtual spaces, such as the desktop and folders, to organise and categorise objects within the computer. In this section, we highlight some universal factors in the modern desktops such as Microsoft Windows, Macintosh Finder, KDE and Gnome.

2.5.1 Names, Icons and Objects

Objects in a desktop environment will typically have an iconic representation. The iconic representation will typically be a visual cue as to the kind of object that is represented. For example, a Word document appears as a sheet of paper with lines of text visible. Iconic representations, or simply "icons", are not unique. They simply represent kinds of things. Icons in the desktop environment are typically named, usually by an icon label shown below the iconic representation. So objects in the desktop environment are visually represented by icons, to represent their behaviour and are named to identify the particular documents. In some desktop environments, behaviour is extracted from a document's name by the form of an "extension". An extension such as ".doc" in Windows will typically cause the document to be opened by Microsoft Word when the icon is opened.

Names in the desktop environment are not unique identifiers. Two icons can share the same name if they reside in different folders. The names in the filesystem may translate directly to files stored in the underlying filesystem. The fully qualified namespace of icons is typically unique. The original reason for this is probably due to the filesystem of many computers. It is important to note that some desktop environments will hide an extension of a document if it understands what that extension means. Although the fully qualified names will differ, the names on screen will be the same. There are other ways to differentiate documents that might share the same name such as location in the virtual space.

2.5.2 Direct Manipulation

The interaction of the user with the desktop environment is known as *direct manipulation*. Direct manipulation is typically performed in the desktop environment with a pointing device, often a mouse, accompanied by a number of buttons that can be pressed whilst pointing. With such a device, a number of actions are commonly used to be performed with the buttons, such as "click", "double click" and "drag". These actions are used to perform tasks such as object acquisition and to move objects about the virtual space offered by the desktop.

2.5.3 Scripting technologies

Many desktop environments offer a scripting ability to reduce the amount of direct manipulation required to perform tasks. The AppleScript language accompanies the Macintosh Finder, and Windows utilises MS-DOS batch files. In particular, the implementation of AppleScript allows the iconic representation of an AppleScript to perform similarly to a Macintosh application icon. A script in a UNIX environment is typically a text file with a special header to signify that it is to be executed by a scripting language. A script that contains the following line will be a script written in the perl [Shapiro 95] language.

```
    #!/usr/local/bin/perl
```

All of these scripting languages are textual. One might argue that there are spatial qualities such as indentation, but the respective interpreters mostly ignore this. In the case of AppleScript, scripts are often stored in binary, and the AppleScript editor performs the visual layout of the script.

2.6 Conclusions

In this chapter, we have examined the role of scripting languages in the modern desktop environment. We noted the power of the UNIX shell and the availability of equivalent languages for the modern client operating systems of Microsoft Windows and Apple MacOS. The requirement of such scripting languages to reduce the amount of direct manipulation in the desktop environment is noted as an important use of such technologies.

We observed the impact of the URL in unifying the remote access services into a convenient string based descriptor, and introduced the IP stack as the method of delivering services over the internet protocol. We discussed the internet browser, and related operating system services, as a means of interaction with the internet.

3. Prograph

3.1 Introduction

In this chapter, we give an informal introduction to Prograph. Prograph is the implementation language of all variations of Psh, as it is a close parallel to our domain of the desktop environment. It is a visual programming language that utilises the dataflow metaphor; a sequence of operations is represented as a directed acyclic graph. Visual nodes depict the operations with arcs (dataflow links) flowing in from the top and out of the bottom of each operation. Each graph contains two special nodes to allow data transfer with the calling graph: the input bar and the output bar. Inputs and outputs on the visual nodes are marked with symbols. Inputs are called *terminals* and outputs are called *roots*. This scheme allows us to build graph networks to describe dataflow algorithms. Each node typically has a textual name that serves as a reference or a label, to cue the programmer as to the purpose of the node. To support conditional programming, Prograph methods are defined as a set of graphs. Each graph is tried in succession until one successfully terminates. Control annotations, similar to those of Prolog, are used to control the flow of execution between graphs.

The development environment of Prograph is interpreted. A project can be compiled to a self contained executable, but this is not required to develop an application, as the interpreted environment is fully capable of executing a project without the need for compilation. The prototypes of Psh that we have built are realised from the interpreted environment, which includes the graph editors and the ability to selectively execute graphs.

Prograph is an object-oriented language. It offers parametric polymorphism via a data directed dispatching mechanism. As Prograph is an untyped language, the resolution of effective methods is determined at method call time by looking up the required method name in the method name space of the received object. This allows the method call to remain untyped, thus allowing the successful resolution of methods, even if a method name is used independently in non-touching sections of the class trees. Objects in Prograph are garbage collected. All data, except for numbers, are boxed with a reference count, which allows for conservative mark-and-sweep garbage collection for cycles. The environment is designed to give the appearance that all data is cloned rather than mutated. However, for efficiency reasons, objects are mutated and list functions operate on shallow copies. The class hierarchies are reflective, allowing for introspection of classes, including their relationship with other classes, and also the explicitly defined attributes and methods of each class. We will utilise this reflection later in chapter 6, where we define our own dispatching mechanism for the Prograph class hierarchies.

In the remainder of this chapter, we examine the semantics of operations and the special annotations of Prograph by examining an implementation of the Quicksort algorithm. This is followed by a discussion on the OO mechanism employed by Prograph with special attention to the dispatching techniques available in the language. Finally, we look at some of the reflective properties of Prograph.

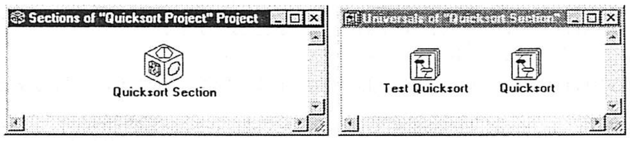

Sections of "Quicksort Project" Project
Quicksort Section
Universals of "Quicksort Section"
Test Quicksort
Quicksort

Figure 2 - A Prograph section, with the universals window opened

A Prograph project consists of a set of sections. Each section is split into three parts: *classes*, *universal methods* and *persistents*. The interactive Prograph environment is based around the editors of these components. The icon that is used to represent sections and section components is shown in Figure 2. The section is displayed as a cube with three visible sides, and clicking a side opens the corresponding editor.

The class editor is used to manage classes. It shows the classes that the section provides to the project, and the hierarchical relationships between classes are shown visually with lines that extend from the bottom of a class to the top of its sub-classes. If the class hierarchy extends over section boundaries, then a *class alias* is introduced as a placeholder for a class so that sub-classing can occur. Two sub editors are utilised to edit classes: the method namespace editor and the attribute editor. The iconic representation of a class is divided into two. The left-hand side opens the attribute editor, and the right-hand side opens the method namespace editor. The class editor is discussed further in section 3.4.

Methods that are not associated with a particular class in Prograph are called universal methods. The editor for the universal method namespace is similar to the class method namespace editor. Persistents are variables whose values are persistent across successive executions of the project.

3.2 Values

The basic types that values can have in Prograph are: integer, real, string, boolean, pointer, external, null, undefined, none, point, rectangle and rgb. The first five are simple datatypes. External types are values that conform to an external typing system (i.e. MacOS or Windows), the last three types, point, rectangle and rgb, are used by the Application Building Classes that are delivered as part of the Prograph distribution. When values are included as constant values in Prograph code, they are specified as shown in Figure 3.

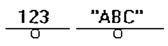


Figure 3 - Constant values in Prograph

When either a terminal (an input to a Prograph method) of an operation or the output bar is not connected, the resultant terminal (i.e. the terminal on the method which contains the loop annotation) assumes the NONE value. The NULL value is used to represent values from a looped repetition that did not complete a single iteration.

3.2.1 Lists

Prograph supports the use of lists by a List type. The list syntax is a sequence of elements enclosed with parentheses, and each list element separated with a space. Examples of Prograph lists are shown in Figure 4. Lists are unrestricted in the types of data that can be stored together in one list. The second list shown in the diagram has an integer, a string and a list as list elements.

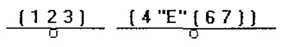


Figure 4 - Examples of Prograph lists

Prograph makes a good attempt to give the impression that it is a pure dataflow language. However, for efficiency reasons, most list operations are shallow.

3.2.2 Objects

Objects are values that flow through dataflow graphs. Again Prograph does not continually clone and destroy objects, but rather reuses them for efficiency reasons. This means that an operation such as changing the value of attributes actually mutates the original object. The syntax of manipulating objects supports the idea of pure dataflow, and a similar language to Prograph could reuse the syntax to implement non-mutable objects.

3.3 Implementation of the Quicksort algorithm

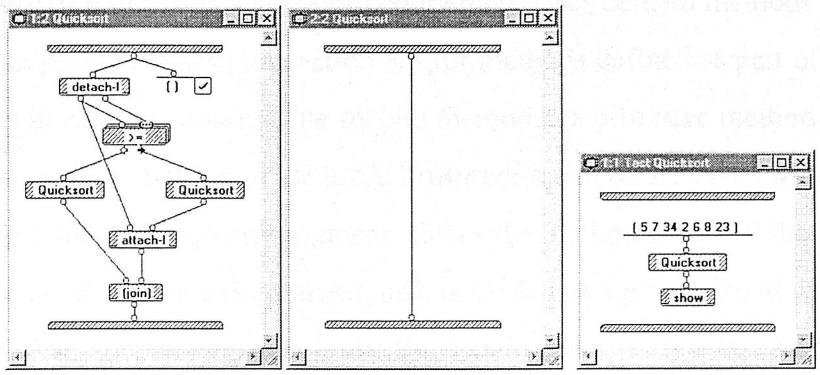


Figure 5 - Implementation of the quicksort algorithm.

The two large method windows on the left in Figure 5 show the implementation the *qsort* method. In general, Prograph methods are built out of a sequence of cases. In the example, the first case is the recursive case and the second case is the base case.

In the recursive case, the first element is detached from the list by the *detach-l* operation. The removed item and the remainder of the list are available on the left and right output roots respectively. The pivot is compared to each element of the list by the next operation.

3.3.1 Method windows

Figure 5 shows the implementation of a recursive quicksort algorithm. The method *Test Quicksort* exhibits a simple dataflow between three entities. For each entity, the small circles, ○, above the entity are input *terminals*, and the circles below the entity are output *roots*. The horizontal bars at the top and bottom of the window pane are the input and output bars respectively. Roots of the input bar are used to obtain data that the operation was called with, and outputs are given to terminals on the output bar. It is important to note that the purpose of executing a window is *not* merely to obtain values for the output bar; if a window successfully finishes execution, then each operation in the window has had the chance of execution.

3.3.2 Operations

The first entity in Figure 6 is a user-defined method. User defined methods which are not defined as part of a class (see section 3.4 for methods defined as part of a class) are known as *universal methods*. The second method is a primitive method (denoted by the two lines at the bottom of the box). Primitive methods are those which are predefined into the Prograph environment. Either the method is part of the inbuilt methods into the Prograph environment, or it is loaded as a precompiled extension when the Prograph application starts. The third method shown is a *local* method. When a Prograph window become complex, it is often desirable to collect together parts of the graph and turn them into a sub windows. Locals can not be directly referenced, so it is a technique used to tidy algorithms which are too complex. The name given to a local is simply a label, and thus can be used to comment on what lies within the local method.

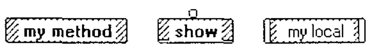


Figure 6 - Example operations. A user defined method, a primitive method and a local method

Primitive methods are methods that are either available as part of the basic Prograph environment, or they are linked in from object code such as that produced by a C compiler.

3.3.3 Control annotations

Dataflow graphs describe the flow of data within a process. To handle conditional computation, Prograph uses *control annotations* to describe process control tasks like skipping to the next case or finishing the current computation. The control annotations are summarised below. The control icons are shown below with '*on failure*' (indicated by the cross in the middle of the icon) and '*on success*' (indicated by the tick in the middle of the icon) control annotations.

Control	Description
☒☑ Next Case	Abort execution of the current case, and start at the next case. If the next case does not exist, then an error occurs.
☒ Continue	Continue execution of the current case. *Continue on success* is the default control for operations, and has no icon.
☒☑ Terminate	Terminate execution of the current case. The output is derived from the results of the last successful repetition. If there was no successful repetition, then NULL values are used for the output.
☒☑ Finish	Finish the execution of the current case. This control causes the current repetition to end. The output for the repetition is derived from the values computed in this case.
⊗⊘ Fail	Abort execution of this operation and propagate a failure to the calling method. The caller has the opportunity to catch the failure by using a control '*on failure*'.

Figure 7 - Summary of the control flow icons for Prograph.

3.3.4 Match operations

The match operation () in the recursive case has a *next case on success* control attached. This means that if the match was successful between the input and the label, then abort this case and continue at the start of the next case. Match operations typically use control annotations to indicate control flow.

3.3.5 Repetitions

The operation in the quicksort example is a repetition. This is denoted by the three dimensional nature, to show that it may be performed zero or more times. Any operation may be transformed to a repetition by introducing a loop, list or partition annotation.

3.3.5.1 Loop annotation

A loop annotation causes a direct relationship between one terminal and one root. The effect is that during successive iterations of the repetition, the value of the terminal is the value that was given to the root in the previous iteration. Terminals which are not looped will provide the same value to each iteration of the loop. An example of the loop annotation is shown in Figure 8.

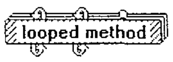


Figure 8 - A Prograph method with loop annotations.

This kind of repetition continues until it is halted by a control annotation within the looped operation. If the loop is *terminated* before any successful iteration of the loop, then the result of the loop annotation is NULL.

3.3.5.2 List annotation

The list annotation, ⁅··⁆, means that an operation will be executed repeatedly with each element of the list in turn, not on the list as a whole. An operation that has a terminal that has the list annotation is forced to be a repetition. The multiplexed operation in the quicksort example tests the pivot (the leftmost element of the input list) with the remainder of the input list.

3.3.5.3 Partition annotation

The partition annotation, ✧ ✦, means that the results from each iteration will be collected into two separate lists. When a partition annotation is applied to an operation, the Prograph editor automatically creates a terminal with a list annotation from which to partition the data. The root of the underlying operation (which must return a Boolean value) is used to determine to which output list a particular item is added. If the test was successful, the result is appended to the ✧ list, and if unsuccessful, it is appended to the ✦ list. Remarkably, the work of testing each item against the sorting pivot and collecting the results respectively is performed by a single operation. The lists produced by the partition are then recursively sorted, and the remaining two operations construct the result.

3.4 Object Orientation

In this section, we introduce the object oriented features of Prograph by a bank account example. First we will define an attribute which will represent the balance of the account, and then two methods *deposit* and *withdraw*, which will deposit and withdraw from the account respectively. Classes may have attributes and methods, and may inherit from another class. Each class may have a constructor, but there is no provision for a destructor, as the Prograph environment is garbage collected. Prograph does not support multiple inheritance nor data hiding.

A Prograph class is split into attributes and methods. The visual representation, ⬡, emphasises this with the attributes on the left and the methods on the right. Inheritance links are shown with arcs between the classes. Each class may have at most one superclass and zero or more subclasses. The arc connects from the bottom of the superclass to the top of the subclass. Each Prograph section has a single class window that contains the classes. Class aliases can be used to create inheritance links that span across different sections.

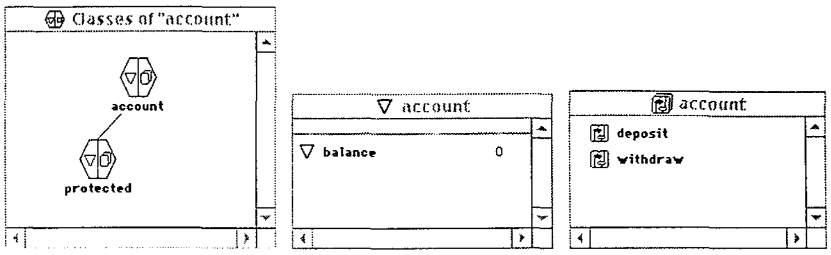


Figure 9 - The Prograph class.

The first window in Figure 9 shows an inheritance link. In this case, the *protected* class is a subclass of the *account* class. Clicking on either side of a class icon opens the attribute and method windows. The remaining windows show the attribute editor and the method editor respectively. Prograph supports two kinds of attributes: *instance attributes* and *class attributes*. The attribute editor of a class contains a green line (shown as grey) with class attributes above the line and instance attributes below the line.

Instance attributes, shown as ∇, have a separate value for each object of the class that they are defined in. The initial value of an attribute in an object is copied from the corresponding attribute in the class definition at the time of object creation. Prograph is a dynamic language, and allows classes to be modified whilst instances of the class exist. For example, instances can reside in the persistent component of sections, or as values of attributes in class definitions. If an attribute is created in the class definition and instances of the class exist, then the attribute is automatically added to the existing instances of the class. Subclasses of an object that contain an instance attribute inherit the attribute and display it with an embedded arrow, ∇, to show that it cannot be renamed nor deleted.

Class attributes, shown as ⬡, are values shared across each instance of the class that it is defined in. The value shown in the attribute editor is the actual value of the attribute. All subclasses of a class containing a class attribute inherit the attribute and display it with the embedded arrow, ⬡, and similar to inherited instance attributes, it cannot be renamed nor deleted. It is important to note that each subclass contains a new shared value for a class attribute and is not shared with the superclass or any subclasses. The Prograph environment contains an option to manually propagate values of class attributes.

3.4.1 Get and Set methods

Figure 10 shows the implementation of the methods of our simple account and the protected account. In the protected account, there is an extra check performed when using the withdraw method in that the resulting balance must not go below zero. The special operations with angled edges are the *set* and *get* operations. These are used to access the attributes of an object. The name of the operation is not a method, but is the name of the attribute to be accessed. In the method *account/deposit*, the first operation is the *get* operation. The first root is a reference to the original object and the second root is the value of the attribute *balance*. The current balance is then summed with the deposit value and then the final operation, *set*, overwrites the existing value of *balance*. It is important to note that the Prograph object model is a departure from the dataflow metaphor. Objects have state. In its purest form, the dataflow metaphor is stateless. Thus we could have achieved the same process by linking the first terminal of the *set* operation to the first root of the input bar.

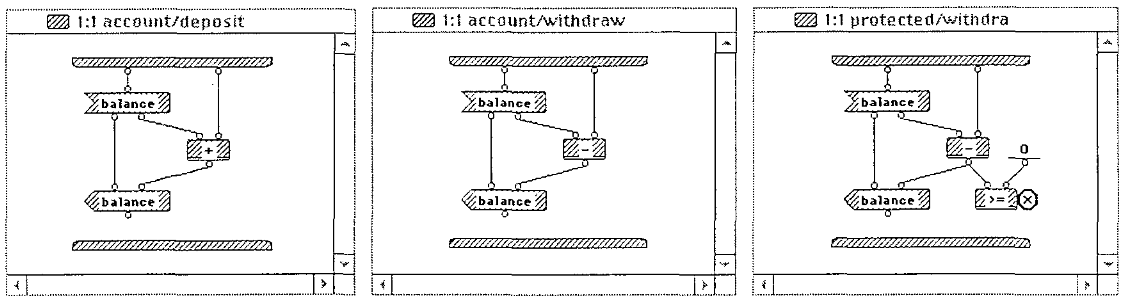


Figure 10 - Implementation of the methods of account and protected classes.

In the *protected/withdraw* method, we make a check that the resulting balance will not drop below zero. If the test fails, then this causes the method itself to fail by using the control annotation *fail on failure*. The calling method has the opportunity to catch this condition by calling the method with any control annotation *on failure*. In Prograph, we can assume that the test is performed before setting the new balance, because the

Prograph system gives higher priority to operations that have a control annotation than those that do not.

3.4.2 Instantiators

The instantiation operator is used to create objects, and is identified by the angled edges on both sides. For a class named *class1*, the instantiator would look like: ⟨class1⟩. The label of the operation indicates the class of the newly created object. The object is delivered on the root of the operation, and a list of pairs can be given to the terminal to pre-initialise the attributes of the object.

3.4.3 Dispatching

Calling a method of an object is similar to calling a universal method. The only difference is that there is at least one forward slash in the operation name. The four notations available are summarised in Figure 11.

Example	Description
/method1	This is data directed dispatching. The method *method1* is called from the object supplied on the leftmost terminal. If the method does not exist, then the class ancestry is searched.
class1/method1	In this case, the class to start searching from is made explicit. Again, if the method does not exist, the class ancestry is searched.

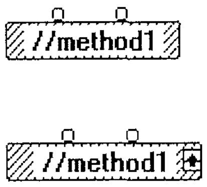
	This notation can only be used in a method belonging to a class. It refers to the class in which the reference itself is made (similar to the `this` keyword in C++). The class ancestry is not searched, but reference can be made to the immediate superclass by use of the *super* annotation. A small upward arrow on the right border of the operation indicates the super annotation.

Figure 11 - Notations of method dispatching in Prograph.

Unlike typed languages such as C++ and Java, the type of data that is used to resolve an effective method is determined during runtime. During the resolution process, the method name is searched for within the given object. This is typically in contrast with typed languages, as the dispatched method is resolved during compile time. With data directed dispatching, the name of the method being dispatched is not tied to a particular class, so the method would successfully dispatch the same function name on two separate class hierarchies. For example, class A and class B have a method named M. An operation with "/M" would successfully find the method for objects of type A and B, even though there is not directly ancestral link between class A and class B. This would not be possible within Java, for example, because the method name would have to be associated with a particular class during class loading time.

3.5 Reflectivity

The object oriented mechanism in Prograph language is reflective. The classes available from within the Prograph environment can be fully examined via a set of Primitives. The classes in the Prograph environment and their relationships can be examined, along with the methods and attributes explicitly defined in each class. The set of primitives that we use in the remainder of this thesis are summarised in Figure 12.

Primitive	Description
type	The type primitive returns as a string the type of the value that it receives.
ancestors	The ancestors primitive returns a list of all ancestors to a particular class. The list is ordered such that the head of the list contains the superclass and each successive list member is the superclass of the preceding list member.
classes	The classes primitive returns a list of all classes available from the Prograph environment.
descendants	The descendants primitive returns a list of all descendants of a particular class.
method-arity	The method-arity class returns the arity of both the terminal and roots of a particular class and method.
method-classes	The method-classes primitive returns a list of classes that explicitly define a particular method.
methods	The methods primitive returns three lists, each containing method names explicitly defined in a particular class. The first root contains the list of normal methods. The second root contains the list of Get methods and the third root contains the list of Set methods. The Set and Get methods constitute an advisory scheme for arbitrated access to virtual attributes.

Figure 12 - Reflective operations

3.6 Conclusions

In this chapter, we have given an introduction to the visual programming language Prograph. It conforms to the dataflow model to execute directed acyclic graphs, and indicates control flow by annotations rather than control flow arcs. We have discussed the values that can flow through dataflow graphs and have discussed examples to demonstrate the language.

Prograph is an object-oriented language, and we have discussed the run-time creation and modification of classes in the Prograph environment. We have discussed the dispatching mechanism, Data Directed Dispatching, and how it obtains parametric polymorphism in a typeless language.

The Prograph environment is a garbage collected environment, and offers reflection of the class hierarchy and the classes. The interpreted environment is as fully capable as the compiled executables that can be produced.

4. Classic Psh

4.1 Introduction

In this chapter, we discuss the early work on Psh as described by [Glaser, Smedley 95]. The reader should note that this chapter is primarily organised to introduce the early Psh model; it is not necessarily a chronological account. Psh as a concept existed prior to this research, however many of the ideas in this chapter were developed in discussion with Hugh Glaser. In this thesis, we refer to the model of Psh that is described in this chapter as "Classic Psh".

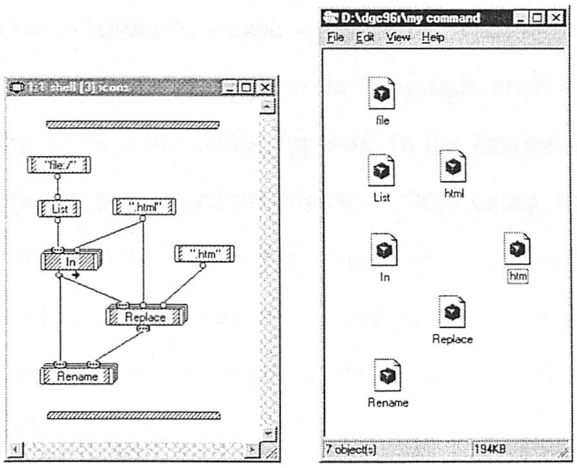


Figure 13 – A Psh window and a windows explorer window.

The development of Psh focused on the similarities and differences of Prograph and the desktop environment of the Macintosh computer, known as the Finder. Prograph is a visual environment for the creation of visual programs, whereas the Finder is a visual environment that allows direct manipulation of files and folders on a Macintosh computer. It was noted that the windows of both systems are rather similar; they both contain a group of iconic objects. We compare a window from Prograph and a window from the Microsoft Windows Explorer in Figure 13, and it can be seen that the only major visual difference is that the Prograph window has directed lines that connect the icons together. In essence, the Finder window contains a set of icons that represent files and folders with little visual programmability, whereas the Prograph window contains a bag of icons that represent a visual program. The goal of Classic Psh was to determine the extent of visual programmability that could be implemented within the desktop metaphor, by producing a prototype in the interpreter environment of Prograph.

In the Prograph environment, there are two actions that we perform. Firstly we can create visual programs by editing the iconic objects, by creating, editing or adding dataflow links between icons. Secondly, we can execute a window of icons to produce results by value or by side effect. The Finder model implements the first action: Editing. The process of editing documents in the Finder is similar to editing constants from within Prograph. Finder documents and Prograph constants are data items that belong to their container; a folder or a case window respectively. Editing is performed by cloning the contents into a relevant editor. In Prograph, such an editor might be a simple edit box for strings, or a list editor for lists. In the Finder, a word processor document would be opened by a word processor. In both cases, the data remains in the container and a clone of the data is available to the editor. If the data is saved, then the original copy is replaced by the version held in the editor. It is this similarity that we claim to be the primary common ground between the existing desktop metaphor and the interactive environment of Prograph.

Extended functionality is introduced into the Finder by introducing Applications and AppleScripts. Applications are special data files that conform to the Macintosh binary format. AppleScripts are textual files that conform to the textual language of AppleScript. Files of either type are treated similarly in the Finder. They can be executed by double clicking on their iconic representation, or documents can be dragged onto them via the Drag-and-Drop gesture to serve as arguments.

In the desktop environment, we need to introduce the concept of dataflow links between icons. We also note the need for a further gesture that is not present in a typical desktop environment. We need the ability to execute a Psh window, which is equivalent to executing the container object. Thus in addition to the ability to execute files on the desktop, we also need the ability to execute *folders*. In terms of the desktop environment, this executes all executable objects in the folder with respect to flow of data described by the dataflow links attributed to the folder.

The remainder of this Chapter is organised as follows: Firstly we discuss our definition of a desktop command: an iconic object, representing a process that can be invoked visually. We also consider the issues involving flags such as those found within UNIX commands. We then discuss the parameterisation of scripts and how this is aided by the dataflow metaphor. An example is given in the form of a word counting example that demonstrates how the *local* mechanism of Prograph is analogous to Folders in the Finder. Finally, we look at a package design to aid in the generation of HTML documents accompanied with comparison to the same package defined textually.

4.2 Elements of Classic Psh

4.2.1 Folders

The desktop metaphor utilises the folder as a means of creating a hierarchy for the documents and applications on a computer. In terms of a filing system, the hierarchy of folders is typically analogous to the underlying directory structure. In the Classic Psh design, folders are represented by the method windows, which appear in the Universal Methods window of a Section, or in the Methods window of a Class. A folder that resides in another folder is represented in the Psh design by the *local* operation. The *execution* of a folder, a new operation to the desktop metaphor, has the effect of executing the contents of the folder; each item in the folder is subsequently executed. Sub folders are executed according to their annotations.

4.2.2 Data files

Modern desktop environments are document centric. We manipulate and categorise documents and perform operations such as editing, printing and mailing documents to other people. In the Psh design, a document that resides in a folder is represented in the Psh environment as a Prograph constant, and has no direct effect when executed.

The existence of a data file in a folder that is executed is similar to that of the Prograph constant; there is no effect other than that subsequent operations can use the value of the constant. In terms of Psh environment, the value of a document is the contents of the document.

4.2.3 Desktop commands

The world that Psh depicts draws on the concept of the desktop command. Desktop commands are special icons on the desktop that perform some action when executed. An example of a desktop command in the current Macintosh Finder would be the printer icon. Dragging a document (providing it as input) onto the printer icon invokes the print method on the document. In this case, the condition for the desktop command to be invoked is that it has received input data. In Prograph, this role of the printer icon could be represented by an operation with a single terminal. Since each operation, inclusive of Prograph annotations, is invoked at most once per case execution, the above condition is held.

Each command in the Psh environment is represented as a Prograph operation. The command can either be invoked directly by the user, or used by another command. Figure 14 shows a simple Psh window, containing three Psh commands. Execution of this window will cause Psh to take the first three lines of one document and write those lines to another document. The **head** function used in the example is similar in functionality to the UNIX counterpart, and it has been implemented graphically using the inbuilt methods of Prograph. The commands **Read File** and **Write File** are part of the file management command suite.

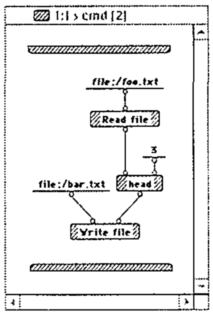


Figure 14 - A simple command that takes the first three lines of the document foo.txt and writes them to the document bar.txt.

The command **Read file** reads the contents of the document `foo.txt`. The content of the document is passed to the **head** command, which generates as output the first three lines of text. In effect, this command writes a file `bar.txt` containing the first three lines of the file `foo.txt`. In a UNIX shell, we might use the following to achieve the same effect:

```
cat foo.txt | head > bar.txt -3
```

4.3 Repetition

The following example illustrates simple repetition. It is a command that renames all the files in a directory with a `.html` extension to `.htm`. The first window in Figure 16 shows the command to rename the HTML files. *grep* produces a list of URLs of the files containing the extension. The following three methods are multiplexed. The ⟦••⟧ annotation on both the input terminal and the output root is equivalent to *map* in functional languages. It applies the operation to each item of the list and collects the results into another list. The remaining input terminal on *join* is a constant value.

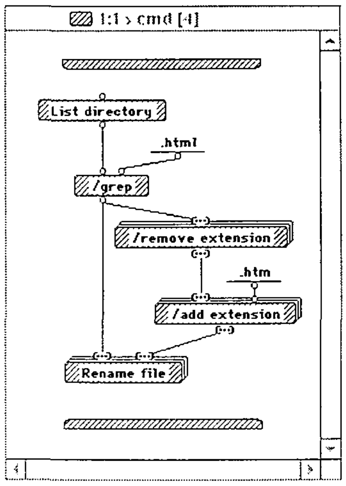


Figure 15 - A command to change suffixes from *html* to *htm*.

4.4 Script Parameterisation

To give a flavour of the facilities available in Psh, we can turn a command into a script by parameterisation. In Psh, a script can receive parameters from the *input bar* (the long horizontal shaded bar at the top of the window pane). In this example, we could allow the user to specify the renaming convention by removing the *.html* and *.htm* constants, and drawing arcs from the input bar to the top of *grep* and *add extension* respectively. In effect, we can develop a command, and place abstractions on it when the need arises. The dataflow metaphor aids the process of parameterisation, as we can directly trace the use of constants.

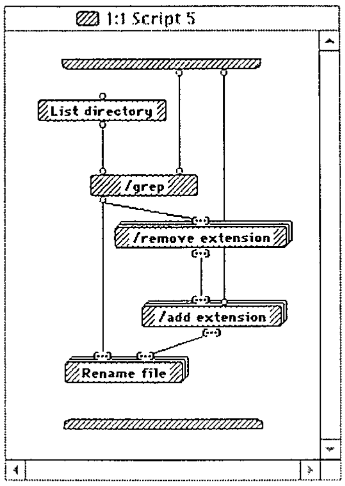


Figure 16 - A parameterised script produced by modifying the command given in Figure 15.

Producing the script in Figure 16 is a simple process of making a copy of the command and taking the parameters off the input bar.

4.5 Example: A Word Count report

In this example, we show the main method of a word count utility, shown in Figure 17.

The first operation prompts the user with a dialogue box containing a string input box. The string is assumed to be a URL, and the contents of the URL are fetched. The document is then processed by *wc*, which has similar functionality to the UNIX equivalent. The results from the *wc* utility are then passed to a *local*. In Psh we regard this as an unnamed script. The label 'gen wc report' is for program readability only. Finally, the results of the report are displayed as rendered HTML. Our implementation of Psh uses Netscape Navigator for this purpose.

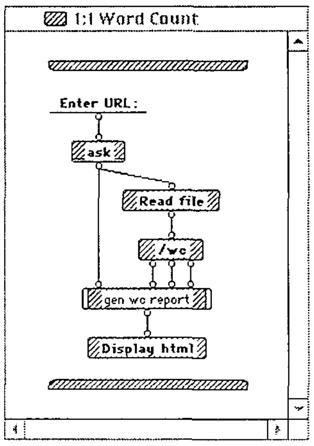


Figure 17 - The main method of the Word Count utility.

It is interesting to note the connections between *wc* and 'gen wc report' in the last example. If we were only interested in the second result of *wc*, we would simply leave the first and third output roots unconnected. It also shows that the notion of multiple return values is easier to visualise in Prograph over traditional UNIX shells.

4.5.1 Flags

Specifying flags on the command line can modify the behaviour of many UNIX commands. A UNIX command flag is typically identified as a word starting with the '-' character, and is sometimes followed by an argument. Flags are often used to change the behaviour of a command without the need for a new command. In UNIX, we would use the *wc* command to perform a "word count" of a document. With *wc*, flags are used to specify which of the following values are to be printed as output: (1) the number of lines, (2) the number of words and (3) the number of characters. Each of these outputs can be independently switched on or off. The default behaviour (i.e. no arguments) of *wc* is to print all three outputs, thus allowing for seven combinations of output. The existence of command flags helps reduce the number of names required in the UNIX command name space.

In the Psh environment, we separate the flags into two classes. The first class describes those that enable or disable parts of the output. For example, we can represent the output of *wc* by three output terminals. If we decide to ignore some of the outputs, the values on the remaining output terminals are not affected. The second class of flags is flags that modify the behaviour. For example, the comparison algorithm in *sort* can be reversed.

It is useful to make this distinction, as Psh commands can produce multiple outputs just as easily as single outputs. In Figure 17, the first two outputs of *wc* are ignored. In a UNIX shell, we would either have to specify a flag with *wc* to select only the character count, or use a separate utility to strip the unwanted values from the output. By providing multiple outputs, we can implement desktop utilities with fewer flags than their UNIX counterparts.

4.5.2 Dataflow oriented html generation

Part of this research involved bringing the desktop close to the World Wide Web. We implemented a simple suite of HTML generation methods that renders text into HTML. For example, turning a block of given text into a paragraph simply involves inserting the HTML tag `<P>` to the start of the text, and appending `</P>` to the end. Figure 18 shows the simplicity of this approach.

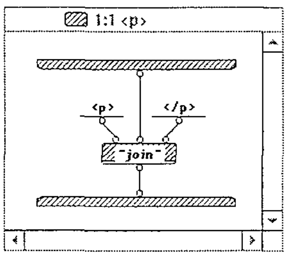


Figure 18 - A simple method to produce an HTML paragraph.

In cases where a tag allows parameters to be set, the method has an extra input terminal to receive the parameters. Leaving the terminal unconnected implies that no parameters are to be specified.

All of the methods in the HTML generation suite have a single output root that yields rendered HTML. The number of input terminals is dependent on the particular tag. For example, the horizontal separator <HR> does not operate on a block of text (and does not have a corresponding </HR> tag), so the only input that it might take would be parameters to the tag itself. In the case of the anchor tag <A>, shown in Figure 19, the tag operates on a section of HTML (it turns it into a hypertext link), and can also be parameterised. The *NONE Check* operation checks if the calling method left the *parameters* input unconnected, and replaces the Prograph NONE datatype with an empty string.

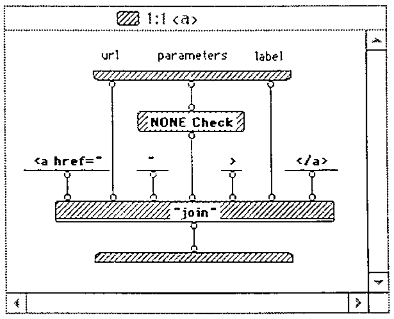
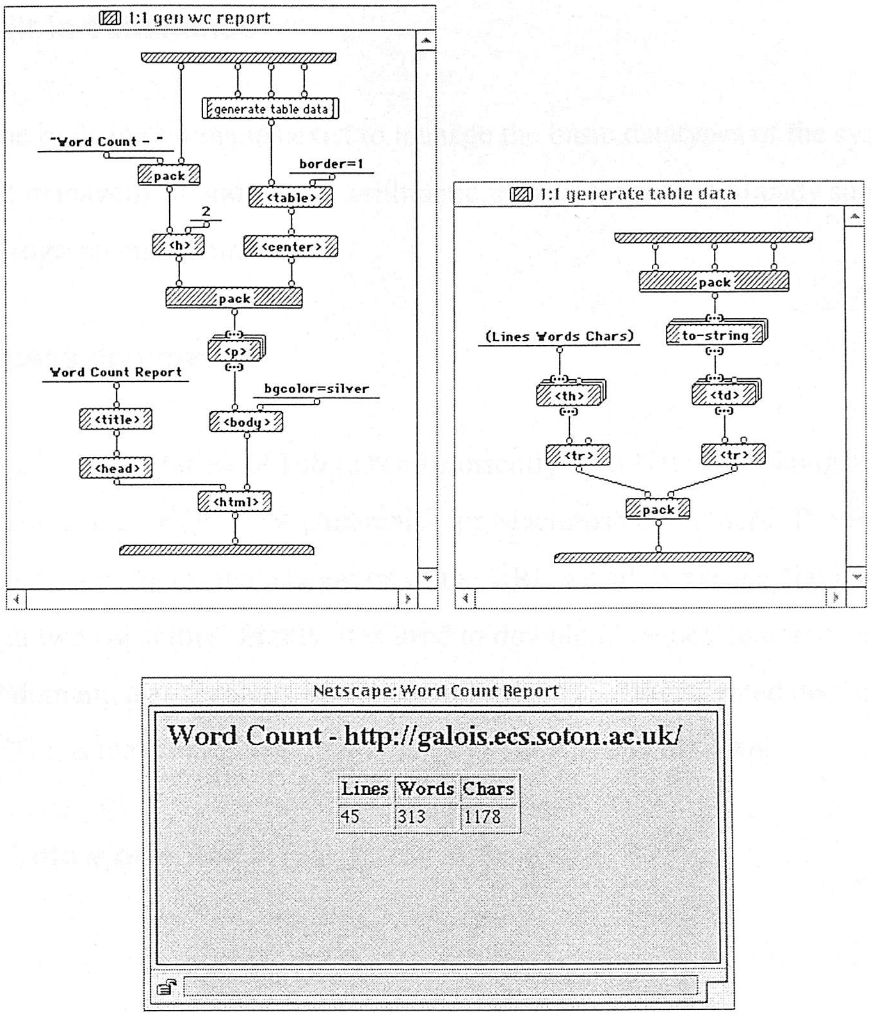


Figure 19 – Implementation of the anchor tag, using a URL, a label, and an optional set of parameters.

The code for generating the html document in the word count example, and the results from an invocation of the script, is shown in Figure 20. Note that the multiplicity of data (i.e., single item, a list of items) is handled by the list annotation.



Netscape: Word Count Report

Word Count - http://galois.ecs.soton.ac.uk/

Lines	Words	Chars
45	313	1178

Figure 20 - Code to produce the word count report document, and an example output document from Netscape.

4.6 Implementation status

A working prototype of Classic Psh has been implemented on the Apple Macintosh operating system, using the Prograph CPX environment. We have integrated these systems with Netscape Communicator and Anarchie (an FTP client) at the level of the AppleEvent communications layer of MacOS.

4.6.1 Built-in commands

Many of the built-in commands exist to manage the basic datatypes of the system, such as list management and integer arithmetic. Most of these are already supplied from the Prograph environment.

4.6.2 Access commands

The current implementation of Psh runs concurrently with Netscape Navigator for HTTP access, and an FTP client (Anarchie) for Macintosh computers. The Psh environment co-ordinates the navigation of the URL world. Netscape Navigator is exploited in two capacities. Firstly, it is used to download requested documents from the HTTP domain, and secondly it is used to display HTML generated documents from Psh. The actual communication is achieved at AppleEvent level.

4.6.3 Dataflow oriented HTML generation suite

We have implemented a subset of HTML in the document generator to demonstrate the simplicity of generating a structured document in a visual dataflow language.

4.7 Conclusions

Programmability already exists in the world of the Finder, but only in the textual form of AppleScript. The Psh system was not an attempt to provide more programming power over AppleScript, but to explore the extent we could cover by remaining in the visual domain. The original Psh technology achieved its goal of a single conceptual model combining the desktop metaphor with scripting ability. To bring the desktop metaphor forward to the Psh model would require the introduction of user-definable desktop machines, dataflow arcs and an extra gesture to *execute* a window.

Prograph has the property that during the execution of a graph, each node is executed once and only once. This is a departure from the UNIX pipeline model, where processes are executed in parallel. At an abstract level, the Psh design is not restricted to either model.

Psh has powerful features over a typical UNIX shell in that we can manipulate multiple return values as easily as single return values. It has cleaner syntax for reusing an output from a node as input to several other nodes, because we can visually see to where the output is flowing. In contrast, the UNIX *tee* command can be used to duplicate the output of a UNIX filter, which is similar to having two dataflow arcs from a node output. The tee command reads from standard input and writes to standard output, but in addition will write the dataflow to a number of specified files. The following example filters out the lines from the file *bar.txt* that do not contain the word *foo*, and counts the number of characters in those lines. As a side effect, the lines which are used to count characters from is written out to the file named *f*. A UNIX pipeline which describes the above behaviour would be:

```
cat bar.txt | grep foo | tee f | wc -c
```

We also discuss the document generation component that generates documents in the HyperText Mark-up Language. The HTML suite can be viewed as a collection of very simple functions, but in the visual dataflow language of Prograph, they become powerful functions that allow the user to express HTML building scripts in a declarative way.

5. DDD Psh

5.1 Introduction

In the previous chapter, we discussed how Classic Psh demonstrated the viability of using Prograph to prototype a programmable desktop interface. In this chapter, we discuss our first attempt at producing an object-oriented version of Psh. The *data directed dispatching* version of Psh, DDD Psh, is the result of applying the native OO mechanism of Prograph to Classic Psh.

An object-oriented paradigm within a typeless language, such as Smalltalk, gives a different feel to typed languages such as C++ or Java. When we refer to a language as typeless, we mean that a reference to a method is untyped. The objects are of course typed, and are dispatched at runtime. In Prograph, the method identifier is simply its name. This differs from typed languages, where a method identifier includes not only the name, but also the class in which the name is defined. If we consider an example where a name *n* appears more than once a set of class hierarchies, we can see different behaviour between typeless and typed languages. A reference to *n* in a typed language is applicable to the class and subclasses of the type of the reference. In an untyped language, all methods named *n* are applicable. This property of the data directed dispatching mechanism allows for more declarative freedom within the scripts that we are able to define in DDD Psh.

In an effort to expand the desktop for internet and intranet use, we have chosen to refer to files via URLs. The *URL* class generalises the access to the resources available to the Psh environment. A currently used small hierarchy is shown in later on in Figure 22. The URL class gives us the freedom to apply Psh scripts on different URL schemes. For example, in the UNIX shell, we could write a simple command to print all documents in a given directory starting with the word `foo`. If that directory is only available via the FTP protocol, then our UNIX shell command becomes non trivial. Such a command is much easier to write in our scheme, since the access class distinguishes between files by URL scheme, and allows us to use *data-direct* *dispatching* to call the relevant implementation of a particular utility. When we examine the target directory, we do not explicitly refer to the mechanism used to carry out the task, we simply apply the operation to retrieve the directory contents to the URL object.

One of our objectives for producing an object oriented version of Psh is to utilise specialisation possibilities provided by producing sub classes of the kinds of data that can be used in the Psh environment. Using classes and class aliases, the Psh environment can be augmented with new functionality by adding Prograph sections. Prograph sections serve well as an extension model for Psh, in much the same way as they are used to extend the capabilities of Prograph.

In this chapter, we discuss the definition of the Data class hierarchy for DDD Psh. The consequences of applying object-oriented design to Psh are discussed using the URL class as an example. Finally, we discuss the concept of coercion within the Psh environment and how the 'lazy' implementation allows us to reduce the number of coercions required in the execution of a script.

5.2 Data class hierarchy

The Psh environment allows users to perform direct manipulation on a number of diverse datatypes, such as text, pictures and sounds. In our object oriented model of Psh, these are represented in the data class hierarchy. Operations which can be performed on the data in the data class hierarchy are implemented as methods within the relevant classes. Some methods will not be applicable to all datatypes, such as *gamma correction* [Foley 90]. Gamma correction is a process which is used to correct image colours so that they are not too dark of bleached out on output devices such as computer displays. In this case, the method will only be defined in the *Image* class and its subclasses. An example population of the *data* class is shown in Figure 21.

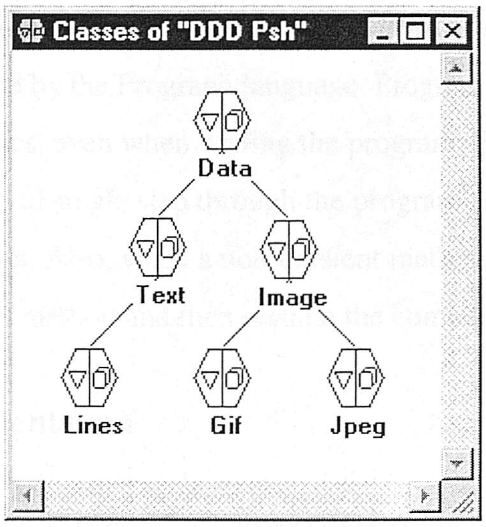


Figure 21 - An example class hierarchy of the data classes.

The envisaged desktop environment contains diverse datatypes, such as pictures and sound. We would expect a good word count package to use techniques, similar to OCR and speech recognition on these types. An example population of the *data* class hierarchy is shown in Figure 21. The example shows a *Text* datatype and also a *Lines* datatype. Many utilities, such as *grep*, process and give output in lines of text.

The data class hierarchy exists to describe the kinds of data that the user can manipulate in the Psh environment. Subclasses in the data class hierarchy can have more than one purpose. If the subclass describes a new raw datatype, for example a JPEG image that is subclassed from the Image class, then we expect the interface to the data to remain similar to the picture class, accompanied by coercion routines to and from the JPEG image format to the internal representation of the Image class.

Pieces of data in the system are actually objects in the Prograph environment. When a command is dispatched on an object, the rules of data directed dispatching, Prograph's dispatching mechanism, are used to resolve the effective implementation. Any method name can be invoked on any object in the system. If the method does not exist, then a run-time error is caused. In a language such as Java, the method must be known to exist before the program will compile. As our "program" is the desktop environment, and our usage of the desktop environment is similar to "building a program", data directed dispatching allows us to keep the desktop in a workable state at all times. This is a concept provided by the Prograph language. Programs remain in an executable state at all times, even when writing the program. For example, we can create an empty method and single step through the program. We can author the graph just ahead of the execution. Also, when a non-existent method is called, we have an opportunity to author that method and then resume the computation.

5.2.1 Access implementation

Turning to the issue of file system access, we note conceptual and technical problems when using URLs as the general resource identifier. Conceptually, the problems relate to giving the user consistent views of local and remote file systems, such as HTTP. Technically, HTTP [Berners-Lee et al. 96] puts constraints on access methods. With URLs, the path of the document and the name of the document itself can specify a document. However, we cannot be certain that we are able to examine the contents of the directory that the document is stored in, because most HTTP servers allow for a "default" page. For example, consider a server that assumes the default document for each directory to be named `index.html`. If a particular directory, `http://www.foo.com/`, contains an `index.html` document, then specifying the directory name will actually return the default document, `http://www.foo.com/index.html`. This masks the contents of the directory away from the client side user, and HTTP/1.1 does not support directory reading by any other method. If a directory listing *is* returned, we are then faced with the problem of parsing it. The "directory" that is returned is a document containing links, and HTTP servers are free to structure this document in their own way. Unfortunately, there is no method to reliably determine if the returned document is a virtual directory listing, a default document, or even a document informing us that the directory does not even exist.

The access to local and remote files from Psh is generalised by the *access* class. This class is specialised with subclasses that correspond to access schemes that are available to the Psh system. Figure 22 shows the subclasses for local file access, remote access via the FTP protocol and the World Wide Web. This class hierarchy provides a common interface to access data, regardless of the underlying transfer protocol. If we wrote a script to process data from a given URL scheme, we could reuse the script to process data from the local file system, or any other URL scheme. Of course this is classic separation, but other systems don't do it. Cardelli [Cardelli 97a] views the World Wide Web as a distributed system. Many people have attempted to classify what the exact nature of the internet is, and there are compelling arguments that claim that the web is a distributed hypertext system, a distributed filing system or a distributed computer. Perhaps we can accept that the internet is just a distributed system. Cardelli examines some of the global structures over the internet as FTP a global file system, Telnet as a global multiprocessor and HTTP as a global document system.

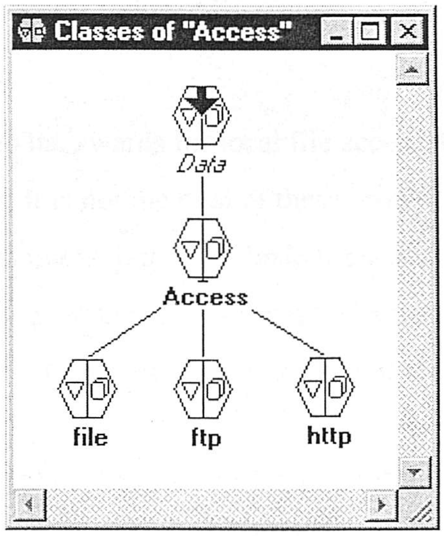


Figure 22 - The structure of the access class hierarchy.

The availability of services that an access scheme offers is defined by the methods available in its class, as shown in Figure 23. Naturally, the classes will support only the methods that are supported by the relevant scheme. For example, deleting a directory using the HTTP protocol is not generally supported, and so the method is not defined within the class. A script which attempts to perform an unsupported operation like this will fail. However, if a particular web server does support deletion of directories, then we might create a subclass of HTTP, containing the extra method.

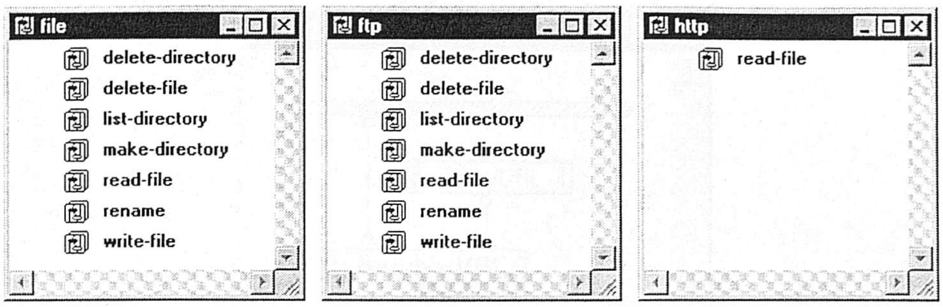


Figure 23 - The method population for the access classes.

There are two reasons that an access method might fail. Firstly, the protocol used might not support the operation. Secondly, aspects of the operating system or remote system might prohibit certain operations. In a networked environment, we should not be able to delete a file belonging to another user, if that user has not given us permission.

Using the URL is not a step backwards for local file access. Our initial use of the URL has been via web browsers. It is not the goal of these browsers to provide a replacement desktop environment, but to navigate the world wide web in a convenient manner. It is the shortcomings of the browser, not the concept of the URL, that restricts using such browsers for general manipulation of the local file system.

The question of why we need URLs in a desktop environment at all when we have iconic representations of files can be raised. Iconic representations are not usefully transportable via mediums such as newspapers, posters or even the side of a cereal packet. We cannot express such images to the computer in a convenient fashion, unlike the URL. Once a URL is entered into the system, it can be resolved to an iconic object that we can manipulate thereafter.

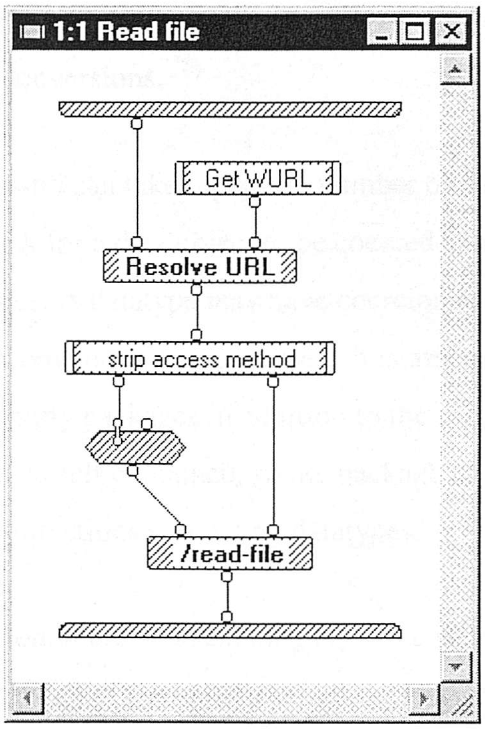


Figure 24 - An example of creating an access object from a URL.

At the user level, URLs are expressed as strings. Figure 24 shows the universal method for Read file. This is the universal method that is used to read files that are specified by a URL. The hexagon shaped box is a constructor operation. The name of the class is usually specified as the name of the operation, but here we use the *inject* annotation to name the operation at execution time. The *Read file* method resolves the given URL with the Current Working URL, and creates a relevant access object. The *strip access method* local is used to pick out the access scheme from the URL for the injected instantiator.

5.2.2 Coercion of data

Operations on data in the Psh environment typically work on data in one form or another. The different kinds of data that can be manipulated are represented within the data class hierarchy. In our JPEG example, leaving the jpeg data as is when bringing a jpeg into the Psh environment fits in the general philosophy of Psh. In a naïve system, we might convert the jpeg directly into the internal representation of the picture class.

Ideally, we would want to leave data in the most convenient form in an attempt to reduce unnecessary data conversions.

Upon invocation, a command can take input of a number of data objects. The command will typically ask for a data object to be coerced to a specific datatype before processing continues. A datatype may have coercion routines to coerce from itself to another type, or from another type to itself. It is assumed that packages are designed to refer to third party packages in addition to the original datatype set of Psh. A package is supposed to be self contained, so the package author can implement coercion routines in both directions to existing datatypes.

During a particular coercion, there is an existing type (the source) and a requested type (the destination). With the web of coercion routines accessible via different packages that have been installed, it is possible that the coercion may be done in one step, multiple steps or not at all. The coercion routes from types to other types can be generalised by a directed graph (not necessarily acyclic). The process of coercion in Psh cannot be implemented completely via a classic tree-style inheritance mechanism. There are possibilities here for disjoint sets of datatypes, which is to be expected.

In the case of a jpeg image, the machine may be equipped with hardware to manipulate jpeg images. If our operation is supported via the hardware, it can be utilised directly. If our operation is not supported, but is supported via the picture class, then the image is converted to the internal representation of the picture class, and the operation is then applied. Conversely, if we have an image of another type other than jpeg, say gif, and we apply an operation that is only supported via the jpeg datatype, then we coerce the gif to the jpeg datatype, and perform the operation. Remember that in Psh, we always copy (or appear to copy) data, not change it in place. This means that our original image is not touched. Although we have now performed our operation on a potentially lossy image, we note that the only other alternative is to not perform the operation at all and fail.

As an example of a different approach, consider the processing of text files. In Figure 21, we have a special kind of text class called *Lines*. This class stores text as a list of lines, so that operations that work on a line by line basis can inter-communicate using a more efficient form. As most UNIX shell utilities process a single datatype, a binary stream, the UNIX utilities often have to convert their output back into a form which is useful for the next utility to process. In the case of a grep followed by *head*, this results in unnecessary conversions taking place, because if *grep* were allowed to leave its output as a list of lines, *head* would only need to return the first n elements of that list. Data-directed dispatching allows us to perform this coercion between datatypes, by dispatching the relevant version of a utility depending on the given datatype. For *grep*, because we might only decide to implement a *grep* for the *Lines* datatype, the *Text* implementation of *grep* would apply a standard coercion routine to *Lines* and leave the output in the *Lines* type. Now, if we run two *grep* utilities in sequence, we avoid converting the data back into a text stream between the two *greps*.

In effect, utilities should not perform conversions after processing their data. Conversions should only occur on demand, via the data-directed dispatching mechanism. In the *grep* example above, we see that the output type remains in the most convenient type. In both cases, the output is of the *Lines* type.

Note that the UNIX shell does appear to perform some data-directed dispatching by reference to file extensions and magic numbers, but the dispatching is performed within the commands and not the shell. Modern Windows performs dispatching on datatype, which is typically derived from the file extension. The Macintosh Finder (prior to MacOS X) dispatches on document type and creator type. When an icon is opened within the Macintosh Finder, it will first look at the document type. It will then try and match the application specified in the *creator* type with applications on the computer. If it finds the application which saved it, it will be opened in that application. As different applications can set their own *creator* type, applications that are saved using one application can be set to be opened by that application when it is loaded. This design provides a more flexible model than a simple mapping from file type to application.

5.3 Conclusions

The Prograph class system has provided us with a mechanism to categorise the kinds of data that can be manipulated in the Psh environment. DDD Psh is an implementation of the Psh environment, using the object oriented semantics given by the Prograph environment. The data directed dispatching mechanism has allowed us to partially implement a method throughout a portion of the class hierarchies, such as the *write-file* method in the *access* hierarchy. This solution most naturally reflects the availability of services given by URL accessible resources. By adding the *access* class tree to the system, we have realised that a complete implementation of each type of operation is not possible, nor required. Scripts only fail when an operation is not supported, or fail as a consequence of the medium.

We also have clarity of separation of the concepts used in accessing local and remote repositories of information. We separate the kinds of access primitives, such as read, write, rename, which is abstracted from the Transfer Protocol used to reach those resources. Some operations are not possible with certain protocols, and this is reflected in the *Access* class hierarchy within DDD Psh.

A main failure of the data directed dispatching version of Psh is that logical packages of functionality are scattered around the class hierarchy. We notice the need to 'patch' existing classes when we introduce new data classes so that the object mechanism operates correctly. Considering the requirement for further functionality to be added by third party packages, this is not the optimal implementation. Our requirements for further functionality by (a) not modifying existing classes or (b) not modifying the interoperability of existing classes are not met.

Coercion of data from one kind to another is well supported. Each command classifies the kind of output that it creates; typically that of the most convenient form. We acknowledge that we are not destroying the original data when we coerce, but actually creating new data. The original data is only destroyed as a consequence of garbage collection.

6. FCD Psh

6.1 Introduction

In this chapter, we introduce our revised shell to include a First Class Dispatcher, which we call FCD Psh. The Classic Psh design has extended the desktop metaphor by the introduction of visual programmability. In the previous chapter, we extended upon the Classic Psh design by introducing a class hierarchy, the *data* class hierarchy, to classify the datatypes that the user can manipulate in the Psh environment. The data directed dispatching mechanism of Prograph allowed us to express scripts in a more declarative manner.

Command objects contain state information, arguments, in the attribute space and implementations available for specific datatypes in the method space. The inclusion of parameters in command objects as state data allows us to deal with customised commands as single entities. The above command object that performs a recursive listing might be placed on the desktop with a descriptive label such as *full listing.* A novice user would be able to use, copy and move the desktop command without requiring the knowledge on how it was built.

If all command implementations existed within the command class hierarchy, then all the datatypes that the command is to support must be known to the command author at the time of writing. To keep our aim of allowing extra functionality to be added with extra Prograph sections, we need to allow for extra methods for certain datatypes to be retrofitted to existing desktop commands. We provide a mechanism within the dispatcher of the command class to defer dispatching to the first datatype in the arguments if an implementation could not be found within the command class. The first datatype, called *primary datatype* hereon, then tries to find an implementation from within its own class. This allows the author of a data class to provide implementations for command names when the command exists already. This collaboration between the command and data class dispatchers gives precedence to the command class, which will defer to the data class if the command class cannot dispatch the method.

We now turn to the issue of first class dispatchers. We consider a system where not only the command and data class have dispatchers that are inherited throughout their respective class hierarchies, but also that any class can define a regular method that overrides the dispatching mechanism. The task of the dispatching mechanism is specifically to resolve the effective method for a given command (method) and a sequence of arguments. This is a departure from the normal approach, multi-methods, to dispatching on more than one object. In multi-methods, effective methods are resolved purely on specificity rules.

In effect, the dispatcher is demoted to a regular method which simply takes a sequence of arguments and returns a sequence of outputs. This is convenient, as we can implement this with ease within most object oriented languages, including Prograph. The native dispatcher is called only to call the initial "dispatch" method.

The need for adapting our own desktop environment to our own needs is an important feature in a desktop. Installing an application, e.g. Microsoft Word, will typically install not only the program itself, but also extensions to the desktop such as new datatypes and operations. In Microsoft Windows, the integration of applications into the desktop environment is such that one can view the document statistics within the properties dialogue box from the Explorer. We can see that modern desktop environments can be heavily influenced by the packages that are installed. Because of this, a failure of an application to correctly provide a service will in effect cause the desktop environment to work incorrectly. If a failure occurs during the installation or removal of an application, then our desktop environment might appear to be defective. Later in this chapter, we discuss the properties of the Psh packages, sets of extended functionality to the Psh environment, which are included in the Psh environment as Prograph sections. We note that references across sections are not into a registry, but are bound each time a section is installed or removed. This gives us the assurance that a package cannot be referred to by dead references via the bindings introduced by the dispatching mechanisms.

We extend our objectives with FCD Psh to consider further the operating environment for the package author and the end user. The current desktop environment, as found on the Macintosh computer, provides a simple visual interface to the computer. We extended upon Classic Psh by introducing the DDD Psh, which allowed us to classify datatypes and the operations available to each datatype. FCD Psh introduces a new datatype, the *command* class, that allows us to bind together a reference to a command and the flags used in that instance. We also introduce a new dispatching mechanism, that allows us to define an implementation of a command in either the command class or the dataclass being operated on. Finally, we discuss a modification to the OO mechanism: the first class dispatcher.

In the remainder of this chapter, we define the command class of FCD Psh, where we are able to bind a reference to a command with a set of parameters. We then discuss the interface between the algorithmic part of the Psh language and the invocations of command objects, and a consideration of lifetime of command objects. The dispatcher of the command class and also the data class dispatcher are shown with a discussion as to the semantics that result from those. We discuss the first class dispatching mechanism and how we have prototyped this from within the Prograph environment. Finally, we consider the implications of the dispatching mechanisms with respect to Psh packages.

6.2 First Class Commands

Classic Psh introduced visual programming to the desktop as a simple extension to the desktop environment. The environment contained commands that were similar to traditional UNIX commands, and to provide flexibility, some of the commands were designed to take flags; a direct parallel to UNIX command line flags. Resolving a command in Classic Psh simply involved looking up the command in a global namespace - the universal method namespace in Prograph. DDD Psh introduced a

Data class hierarchy to classify the kinds of data that the Psh environment could manipulate. A command in this iteration of the Psh environment was resolved by looking up the name of the command in the class method namespace of the primary datatype. This allowed the Psh system to dispatch a relevant implementation of a command on the primary datatype. If a command / datatype combination did not make sense, then the command name was absent from the datatype's method name space. In both the Classic and DDD versions of Psh, the reference to a command was by name. Thus specialised versions of commands, those with flags, required a terminal on the command to accept a string of flags.

To apply a specialised command, we would have two choices. Firstly, we might have invoked the command directly, with a flag string attached. If we extrapolate this approach with our previous examples, we might find that the appearance of flag strings would reduce the visual appeal of the script. If we were to invoke such a command via direct manipulation, we would have to specify the flags upon each invocation. This would lead to more effort for the user than we would like. The second option available to us is to create a new command that simply calls the original command, but with the flag string hardcoded. This approach would solve the two problems in our first option, but this presents us with an alternate problem. For each new command created for this purpose, it extends the global command namespace in the same way as any other new command. We might find that as our script library grows, the global command namespace is filled with many variants of already existing commands. A further problem to this approach is that all of the scripts share the same command namespace; care would need to be taken to avoid name conflicts of all the specialised commands.

In UNIX, the scope of these problems is reduced by the shell itself. Each shell session maintains a list of *aliases*. An alias definition creates a new command in the command namespace of the current shell. The scope of such commands is restricted to subsequent command invocations within the same shell. The alias is resolved by simple string substitution. The first word of a command line is searched against a dictionary of aliases. If a match occurs, the first word is replaced with the alias definition. After the following example of an alias definition, we would expect the command "`print report.ps`" to be translated to "`enscript -2r report.ps`".

```
    alias print (enscript -2r)
```

6.2.1 Command Class Hierarchy

In our current iteration of the Psh, FCD Psh, we introduce the Command class hierarchy. An object of a command class represents a reference to a command, the implementation or implementations of the command reside within the class definition itself. A command object is analogous to a textual reference to a command in a UNIX script. A reference such as "`ls -R`" in a UNIX script would be equivalent to an object of the command *List* with the recursive flag set in the attribute space of that object. Command objects persist in the system whilst they can be invoked. If an object is part of a script that is stored away, then the object is stored within that script. If the iconic representation of the command object resides on the desktop, then the object belongs to the desktop and is serialised rather then deleted and recreated between desktop sessions. The Command class classifies the kinds of commands that the Psh environment can invoke, by direct manipulation or within a script, in the Psh environment. Command class objects are equivalent to references to commands in UNIX, i.e. their name, and the flags used for that reference.

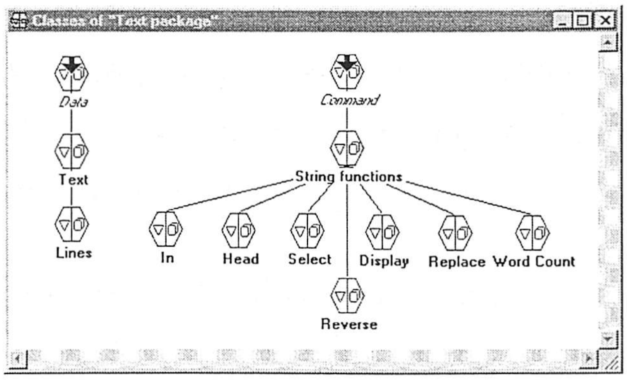


Figure 25 - An example class hierarchy of the command classes.

The commands that are issued in the Psh environment are similar to those used in typical command line environments, such as the example in Figure 25. In scripts, we invoke commands with a sequence of arguments, much like a textual command line. If our textual script is stored away on the local file system, then references to commands persist within the stored script. The command objects in a FCD Psh script persist in the same way. Whilst we have the script available from the local file system, the FCD Psh script retains the command object that was used to build the script initially.

In most object oriented languages, there is a clear distinction between the algorithmic components and the OO components of a language. Systems such as CLOS, an OO extension to common lisp [Steele 90], are typical in that there is a clear distinction between regular lisp constructs and the OO constructs. The dispatcher is a mechanism that is called from the algorithmic part and utilises the class hierarchy to determine an effective method based on an object or objects that are part of the class tree. It is important to note that the callee side of the dispatcher is not necessarily OO, but the called side is. So our intention is to make the placeholder of the dispatcher for a class be available as a method of that class. In our example, we will call this potential method "Dispatch". Like other methods, a class inherits the dispatcher from the superclass if no definition is found within that class.

6.2.1.1 Cases For Failure

Most cases for failure of a Prograph Command are performed either by the lack of a method in the target namespaces or via a Prograph Failure control annotation. At the time of writing, the Prograph language does not support a classic exception mechanism. However, such a mechanism is planned for the language and this would allow for more specific reasons for failure to execute a command or script.

The current mechanism utilises the success/failure condition that each Prograph operation returns in addition to any roots belonging to the operation. Should a script utilise a control annotation upon failure, the control will trigger on any kind of failure. The case where an implementation of a command/data pair is non-existent is treated similarly to the case where the implementation does exist, but failed to execute.

6.2.1.2 Command Factories

Suppose that we want to build a customised command, that is similar to an existing command, but using a different set of parameters. In the Unix shell, we might use the *alias* feature to set the required flags to be used every time the alias is used. In the DDD Psh model, we would have to add a new method to every class in the *data* classes that the customised command is applicable to. This is more work than necessary. The purpose of the *command* classes is to be able to create an instance of a command, with its own set of parameters, which can then be applied to any suitable datatypes. Since the command itself is an object, it can thus be stored away for future use.

It is worth noting that we have two options available when we create a customised command. We can either create a new subclass in the class hierarchy, or we can instantiate an object and modify its parameters to suit. By creating a new subclass, we effectively create a new 'factory' for the customised command which might be further customised to create new factories. By modifying an existing desktop command, we create a 'one off'. This might be preferable, as it would avoid populating the command class hierarchy, our toolbox of commands, with many one-off desktop commands.

6.2.1.3 Persistence of command objects

The lifetime of a desktop command is shown in essence in Figure 26. Section 1 shows that we are making a new desktop command of the type *Select*. In our desktop environment, this would cause a new command icon to appear. Section 2 shows an application of the command. In this instance, the command still holds the default set of attributes that it was created with. In section 3, the command's attributes are modified. In this example, we are changing the output of this particular *Select* command to output lines that do not contain the search word. Section 4 shows the command being applied with the new set of attributes.

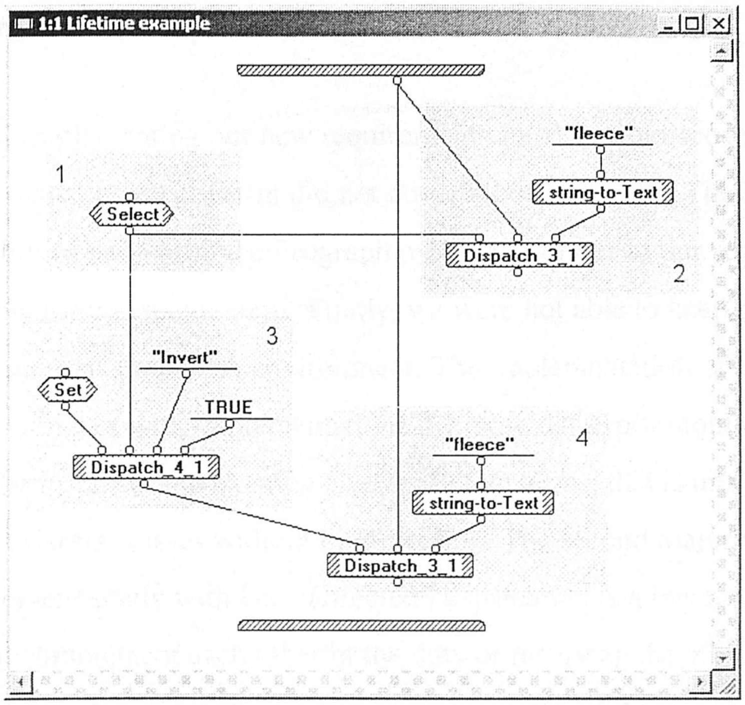


Figure 26 - Example showing the lifetime of a command.

This sequence of events models the actions a user might use to customise a UNIX shell environment. The user in this case has identified a regular need for a command that is achieved by modifying an existing command. In the UNIX shell, the user might create a script, or create an *alias*. The example shown is equivalent to an alias. In a UNIX shell, it could be achieved with the following:

```
alias mygrep grep
mygrep fleece | more
alias mygrep grep -v
mygrep fleece | more
```

In the example above, we define an alias in line 1 and use that alias in line 2. We then overwrite the alias the definition of the alias in line 3 and use the new definition in line 4. As the Prograph environment is garbage collected, the command object will persist until there are no references left pointing towards the command.

6.3 Command Class Dispatcher

In the course of implementing our new requirements for Psh, we discovered that the existing OO dispatching mechanism did not cover all of our needs. The Data Directed Dispatching allowed us to push the Prograph prototype further to our requirements, but did not allow for two major steps. Firstly, we were not able to keep packages as self-contained sections in the Psh environment. The implementation of a package that extends existing classes with implementations for more datatypes requires the method or methods to be installed into existing classes. In OO terms, this is unsatisfactory; we want to utilise existing classes without modification. The second major step that we could not implement solely with Data Directed Dispatching, is a mechanism where two classes can complement each other in the duty of resolving the effective method. The Prograph dispatcher will only dispatch on a single object. Multi methods, such as those found in Common Lisp, would allow us to dispatch on multiple objects, but the respective classes cannot algorithmically assist in the dispatching process.

It became clear that we required a different dispatching model to the one that Prograph provides. We chose to implement our own dispatching mechanism, which is available as a set of universal methods. The reason for the set is due to the lack of variable terminal/root support for non-primitive methods. We have chosen a naming convention for dispatching. If we wanted to invoke a command accompanied by two arguments and returns two results, then we would use the universal method "`Dispatch_3_2`". The reason for the "3" is that the desktop command object itself is included in the number of given objects to the dispatcher. A few implementations of the Dispatch method are shown in Figure 27. Note that if Prograph had a first class dispatcher, we would not need the Dispatch methods. By a first class dispatcher, I mean a mechanism to intervene in the process of resolution of effective methods using the language's normal object oriented annotations.

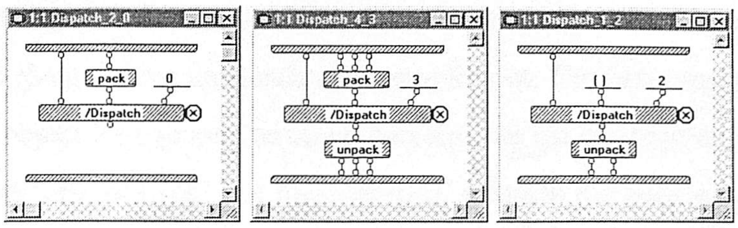


Figure 27 - Wrappers for the first class dispatching mechanism

The goal of the Command class dispatcher is to resolve the effective method that is applicable to a particular command C and its primary datatype D. There are two locations where the implementation of a C/D pair may be found. Firstly the class of C can offer an implementation for D. In our prototype, we have named such methods as "comm *d*", where *d* is the name of the primary datatype. If such a method does not exist within the namespace of class C, we attempt to find the effective method by testing the second location. We examine the namespace of class D for an implementation of class C. Such a method is named "data *c*", where *c* is the name of the invoked command. So we can see that for a given a pair of classes C and D, the effective method can be defined in either C or D. The existence of an implementation in C will override an implementation in D. A special case exists if we consider a command with no arguments at all. As there is no primary datatype in such a command, an implementation can only exist in the namespace of the command. To avoid the trailing space in the above naming convention, the method name of the implementation is simply "comm".

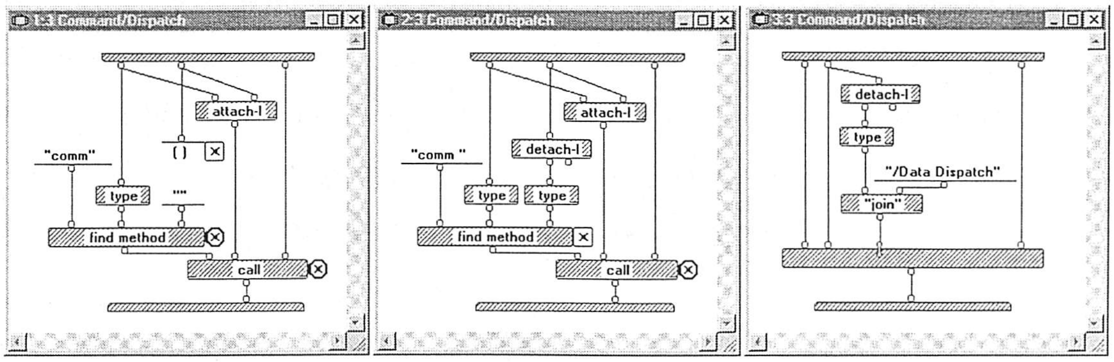


Figure 28 - The Command Class Dispatcher

The implementation of the Command class dispatcher is shown in Figure 28. The Dispatch operation has three terminals and a single root. The first terminal contains the Command object. The second terminal contains the list of arguments; the head of the list is the primary datatype. The third terminal refers to the number of roots expected from the effective method. The implementation of the dispatcher contains the Prograph primitive “call”, which allows us to specify the terminals and roots of the called method as lists. However, if we use this primitive to call Prograph primitives, which can have an arbitrary number of roots, Prograph needs to know beforehand exactly how many roots are expected. If we were to call “detach-l” to remove items from the head of a list, we would receive a new list containing items from the head of the original list. The third terminal of the “call” primitive determines the length of the new list.

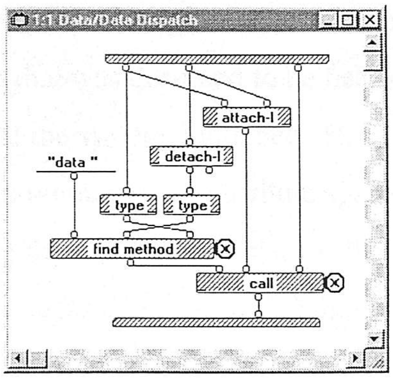


Figure 29 - The Data Class Dispatcher

The implementation of the Data class dispatcher in Figure 29 shows a similar process where the primary data class tries to find an implementation with the name “data *x*”, where *x* is the name of the originating command class. Here we note that if an implementation is not found, then the whole dispatching operation fails. This can then be used to trigger control annotations in the calling method.

6.4 First class dispatching mechanism

Our investigation into a suitable dispatching model led us contemplate an OO model that allowed first class access to the dispatching mechanism. We now consider the issues of allowing each class the choice of overriding the class's dispatcher. A survey of first class dispatchers reveals that the dispatcher is typically built with a view to be hidden away in the final OO model. An exception to this is CLOS (Common Lisp Object System), where we can define a new dispatcher in a meta-class. It is not documented whether objects of different meta-classes can be used in conjunction to each other. In practise, this is not a widely used facility. What is known is that a class with a custom dispatcher is defined by creating a meta-class with the custom dispatcher. I propose to simply make the dispatcher available to be overridden as any other regular methods of a class.

This proposal stems from the OO mechanism present in the AmigaOS 3.1 operating system [Amiga]. The first versions of the operating system were split into modules (shared libraries) that behaved similarly to objects, except that only one instance of a library could be present at one time. AmigaOS 2.0 and later versions contained a system wide OO mechanism that was designed to be backwardly compatible with the original C-type structures that the system contained. The system was implemented so that a pointer to an object is a pointer to the attribute space of the object. The object housekeeping data (including the vector to the class's dispatcher) is stored with a negative offset to the object's data. The kind of OO that we are about to introduce is a refinement of this scheme. We employ a similar scheme where a dispatcher is optionally specified. If a dispatcher is not specified, then the dispatcher from the superclass is inherited.

If a subclass is unable to perform a request that the superclass can perform, then typical inheritance models prevent us from "removing the method" from the subclass's method namespace. Since Prograph employs data directed dispatching, we can attempt to call methods whether they exist or not. Because of this, it would be possible to remove methods from particular classes, as the semantics of the language can cope with non-existent methods. In languages where we have the ability to implement our own dispatcher for a class, we are able to cope with this by either filtering out the specific methods that are unavailable or by failing to recognise unknown method names.

Within the domain of Psh, there are a number of goals to achieve. We would like to be able to treat the namespace of the class methods as commands which are applicable to to particular datatype, thus if a method in unimplementable in a subclass, then the name will not be available to use. We would also like to be able to author classes which can extend onto existing desktop commands without having to modify the existing classes. We have a number of options: There are a number of solutions in the context of Psh:

(a) We implement a feature in the existing dispatcher so that a method can "hide" existing methods. This solution would be problematic in static typed languages, such as C++ or Java, because a variable that is typed to a particular class will allow that variable to be assigned to an object of that class or a subclass of that class. There is typically no mechanism to either hide methods, or catch cases where methods have been hidden. A solution utilising an inheritance model that allows name hiding would be a departure from the classic OO inheritance model.

(b) When a subclass is unable to perform an action of the superclass, we could simply override the relevant methods with simple methods that simply fail (using a Prograph Failure annotation). This is the behaviour we wish the system to produce in such a case, but it requires the class author to know all methods of the superclass in which the subclass cannot perform. If the superclass is changed in the meantime, and a new method is added, then the subclass is unable to predict that case. What we would really like is an inheritance model that allows us to implement the methods that we can implement, and ask the dispatcher to hide all other names from the superclass's namespace (i.e. deny existence of method by default, in dispatcher terms). One could argue that we have a different class altogether, but the classification may be more important than re-using the implementation of the superclass.

(c) To circumvent this problem, we might try to coerce our data object to that of the superclass, and call the method from there. This sounds like a convenient solution to the problem, but if there was a reason why the subclass was unable to perform the method, then there was probably a good reason.

We consider an example in classic OO methodology. Let us talk about a base class *b* and a sub-class *s*. If method *m* is called on *b*, then any extended behaviour of *s* does not affect the ability of *b* to perform method *m*. This is expected behaviour. If method *m* is called on *s*, and *s* explicitly supports *m*, then *m* is called from *s* with the attribute space of *s*. If method *m* is called on *s*, and *m* is not explicitly supported from *s*, then *m* is called from *b* with the attribute space of *s*. Such an inheritance model requires that a sub-class such as *s* should be aware of the interface of *b* in terms of both the public methods and the public attribute space. In classic OO methodology, if we modify the interface of *b*, then it is not certain that *s* will behave in an appropriate manner. *s* will inherit the new behaviour, even if it is a simple case of new functionality rather than a modification to existing functionality.

To demonstrate this, we consider a possible arrangement within the Data class hierarchy that the Picture class and its children describe the kind of pictures that the Psh environment can manipulate. Figure 30 shows the Picture class with two sub-classes, gif and jpeg. The gif datatype is part of the original environment and the jpeg support has been added with an extra package. In keeping with our concept of leaving data in their regular form, the internal representation of a class such a jpeg would be the data in jpeg form. The picture would not be converted to the Picture class representation upon recognition. Thus if we were to recognise data as a jpeg image, use a method that a jpeg implementation exists for, then no intermediate coercing occurs. If an operation is requested on a jpeg that is not explicitly supported by the jpeg class, we would normally have to "catch" the request by implementing the method in the jpeg class so that it would coerce the internal representation to that of the picture class and then reissue the request. Now assume that the picture class has been updated to contain a new method, say gamma correction. The jpeg or the gif class are unaware of this new method and would normally have to also be updated to include a wrapper for the **gamma correction** command. One might say that this is an incorrect use of OO methodology to re-use an attribute for a different purpose. In our case, we are classifying data, where each Data class can potentially have a different representation to its superclass. Now if we have access to the dispatcher, we can modify the behaviour of the dispatcher so that in the case of jpeg, when an unknown method is requested, we coerce the internal representation to that of the picture class and reissue the request.

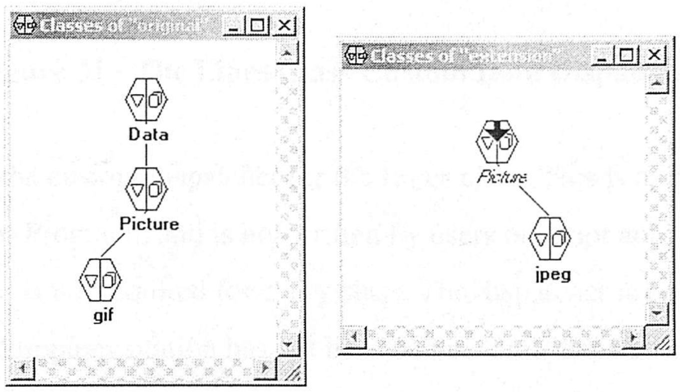


Figure 30 - Example data class hierarchy for pictures

Extra modules of functionality can be added to the Psh environment by inserting Prograph sections, containing classes which are added to the environment. Class aliases are used to identify already existing super classes to inherit from. The new commands and data classes can offer new commands and specific implementations of commands for given data types.

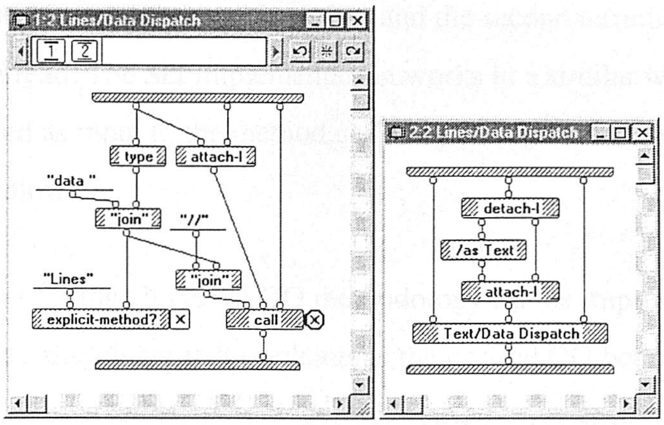


Figure 31 - The Lines Class Custom Data Dispatcher

Figure 31 shows the custom dispatcher for the Lines class. This is a concrete implementation in Prograph, and is not written by users or script authors, and a custom dispatcher is not required for every class. The dispatcher is typically called when a command implementation has not been found. This dispatcher tries to find an implementation within the Lines class. If an implementation does not exist, then it coerces the data to a Text class and re-dispatches the request. In this example, we note that new methods that are defined for the Text class do not need to be explicitly supported by the Lines class.

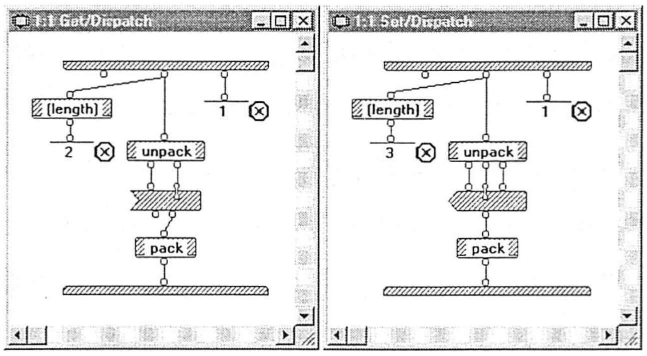


Figure 32 - The Get and Set commands

In Figure 32, we show the implementation of the Psh environment level **Get** and **Set** commands. The implementation of **Get** and **Set** demonstrate some of the power that can be achieved by compounding Prograph annotations. In the **Get** implementation, we can see a get operation with an inject annotation. The first terminal of the get operation is the object that we are to examine, and the second terminal is the name of the attribute to be read. The **Set** implementation works in a similar way. An injected terminal is not used as input to the method called by the operation, and the position of the injection is arbitrary.

We can show that existing classes of OO methodology can be implemented by a root class that contains a dispatcher that implements the desired OO behaviour. To retain the OO properties of that OO methodology, we would simply ensure that the descendants of the OO defining class do not override the dispatcher.

6.5 Implementation status

A prototype of FCD Psh has been implemented on the Microsoft Windows platform using the Win32 version of Prograph CPX. Rather than to re-implement the network support for Windows, we transferred FCD Psh back to the MacOS to provide a full prototype of FCD Psh. Extra functionality can be added by inserting new Prograph sections.

6.6 Conclusions

In Prograph, the classes in a section can be subclassed from a class alias, which refers to a class already contained within the system. This allows us to enrich the datatypes, commands to manipulate and coerce those datatypes with those already in existence, and if necessary, extra transport protocols that might exist for those types. The universals section may contain pre-prepared special versions of the new commands. The persistents section acts similarly as a registry. The Prograph *Section* is a useful mechanism, as it keeps all of this together in a single module that can be installed or removed simply by adding or removing the section from the Psh environment.

In this chapter, we have described first refinement of the object oriented Psh model. The initial OO model, DDD Psh, was implemented using the inbuilt OO mechanism supplied by the Prograph language; characterised by the data directed dispatching mechanism. During the course of developing DDD Psh, we needed to modify existing classes to allow two way coercion between the existing and newly developed classes. This became notably apparent when we added the new functionality via Prograph Sections. Our efforts to redefine the existing classes to allow this kind of reuse were unsuccessful.

The design of FCD Psh focused on the resulting dysfunctional implementation. The classes were authored using our objectives: non-modifiable classes and mutually coercible data structures. The OO mechanism of Prograph would not support the classes that we had defined, so we designed and implemented a new OO methodology that would. The most prominent feature of the new OO mechanism is the first class dispatcher. Upon initial investigation, the dispatcher appeared to be an implementation of multi-methods. But it is not. It is similar insofar that the resolution of effect methods depends on the type of more than one object. How they differ is that the classes involved have some say in the resolution process, allowing us to specify a richer set of dispatching rules rather than a single set of specifity rules.

Initially, we assumed that this mechanism might break some of the important qualities of object oriented systems. However, we found that we can classify existing OO systems and even intermix them to some extent without breaking some of the primary OO concepts. The requirement that a class depends only on its ancestry for resolution of effective methods still holds. Thus dependence on a non-ancestral class only occurs when the class contains a direct reference to such a class.

Essentially, the resultant OO system reverts back to the style of dynamic dispatching (before the static typed treatment). The difference is that a class is capable of supplying it's own dispatcher as a replacement of the one that would normally be used. If one is not supplied, then the one that is effective for the superclass is used. This technique allows us to modify the OO behaviour of classes within the Psh environment.

7. Direct Manipulation

7.1 Introduction

In this chapter, we discuss our vision of how the user interacts with Psh. The Psh technology provides an abstraction of an iconic direct manipulation environment, and it embodies the concepts of manipulation of data via first class data and command objects. We consider how this system appears to the user of a Psh environment, and concentrate on the manipulation of icons rather than the relationship of icons and their containers.

In current desktop environments, two primary mouse gestures are utilised: Point-and-Click and Drag-and-Drop. Point-and-Click is generally used to select items, or to use a contextual menu of an object that is visible. The Drag-and-Drop gesture is generally used to move or copy objects in the desktop environment. We will give state transition diagrams for both gestures accompanied by a discussion on the communication required between the operating system and a client that accepts these gestures.

In particular we focus on Drag-and-Drop, and the usability problems associated with this gesture. We refer to a number of studies that show increased mental stress upon the user and an increased error-rate in a direct manipulation environment caused by the Drag-and-Drop gesture. We note the introduction of 'sticky' mouse gestures to reduce errors caused by muscular tension.

In our investigation into a possible 'sticky' Drag-and-Drop mechanism, we were lead to propose a new gesture: Drop-and-Catch. We compare the state-transition diagrams and client communication of this new gesture and how it achieves two things: it eliminates the need to keep pressure applied to the mouse button whilst performing target acquisition and, more importantly in the context of the Psh environment, gives an increased perception of the concept of dataflow.

We take our design a step further by considering a possible mechanism to invoke commands with multiple terminals and roots via direct manipulation by allowing the user to handle multiple objects simultaneously. Finally, we discuss the history mechanism of the Psh environment and how the visual representation of the history window matches the visual objects that are manipulated to produce each history item.

We implement a demonstrator for the Drop-and-Catch gesture and perform a usability trial on ten users. We discuss the feedback from the users and highlight issues for further research into the Drop-and-Catch gesture and the implications for the desktop environment that it is used in.

7.2 Pointer device gestures

Modern desktop interfaces utilise the pointing device to achieve several tasks, such as selecting, opening, executing, copying, moving and aliasing icons. In the next two sections we shall cover the two primary gestures that are used in the desktop metaphor: Point-and-Click and Drag-and-Drop.

7.2.1 Point-and-Click

Point-and-Click is a target acquisition operation. The user moves the pointer to an object and performs a press and release action on the device button. Objects that can be acquired include icons on the desktop, buttons within dialog boxes, and many more types of common GUI elements. Point-and-Click, in both single and "double click" varieties, is the most prominent gesture used within GUI interfaces that utilise such an input device.

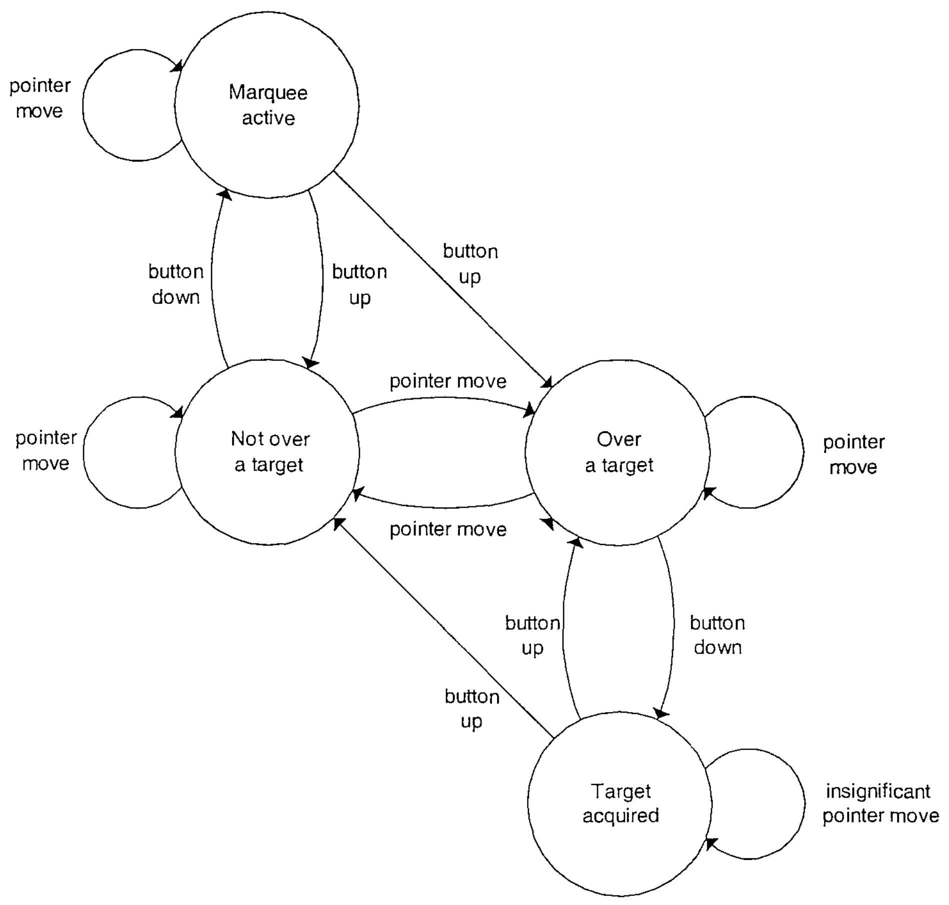


Figure 33 - State transition diagram for the Point-and-Click gesture

Figure 33 shows the state transitions that occur during the marquee drawing and Point-and-Click gesture. The state changes within the input system during Point-and-Click simply involve two events: *button down* and *button up*. The two states shown as "Not over a target" and "Over a target" signify the two states where the pointer button is not pressed, and the state switches between the two as targets are encountered. There are cases where the state can toggle twice and a Point-and-Click gesture is not generated. Firstly, a relevant object might not have been targeted, or the pointer might have moved "too far" whilst the button was pressed. The purpose of the second case for failure leads onto the next gesture, Drag-and-Drop, and is described in the following section.

A single point and click is a gesture involving the acquisition of at most one icon. During a successful gesture, the pointing device is moved to an icon on the display and the button is pressed and released. In an environment such as Microsoft Windows, the Point-and-Click gesture is utilised to perform the *selection* process. Because more than one icon can appear in a selection, the Point-and-Click gesture can be augmented with a qualifier key (in this case, the *Control* key) to add and remove icons from the selection. Once a group of icons has been selected, the user can manipulate them as a group.

In the state transition diagrams, we consider the following aspects of the system: proximity of the pointer with an icon or an icon container, the state of the pointing device button and the presence of icon imagery attached to the pointer.

7.2.2 Drag-and-Drop

The Drag-and-Drop gesture is achieved by a similar process to Point-and-Click, except that during the button pressed state, the mouse pointer is *significantly* moved, or "*dragged*". The state transition diagram depicting this gesture is shown in Figure 34. The Drag-and-Drop gesture is typically used to specify operations that relate to more than one object. Dragging an object gives the user the impression that they have actually picked up an object and can place it in another location. Drag-and-Drop is a common place gesture, used on Microsoft Windows, MacOS and the Common Desktop Environment [Wagner, Curran, O'Brien 95].

Moving icons is intuitive with this gesture. However, it can also be used to copy the object to another location. The two operations, *move* and *copy*, are often achieved by exactly the same gesture. The actual operation performed usually depends on whether the operating system can change the absolute name of the object without actually having to move the underlying representation. In the case of the filesystem, dragging a file from one folder to another folder in the same volume will typically invoke a *move* operation. If the file is dragged across volumes, then the *copy* operation will typically be invoked.

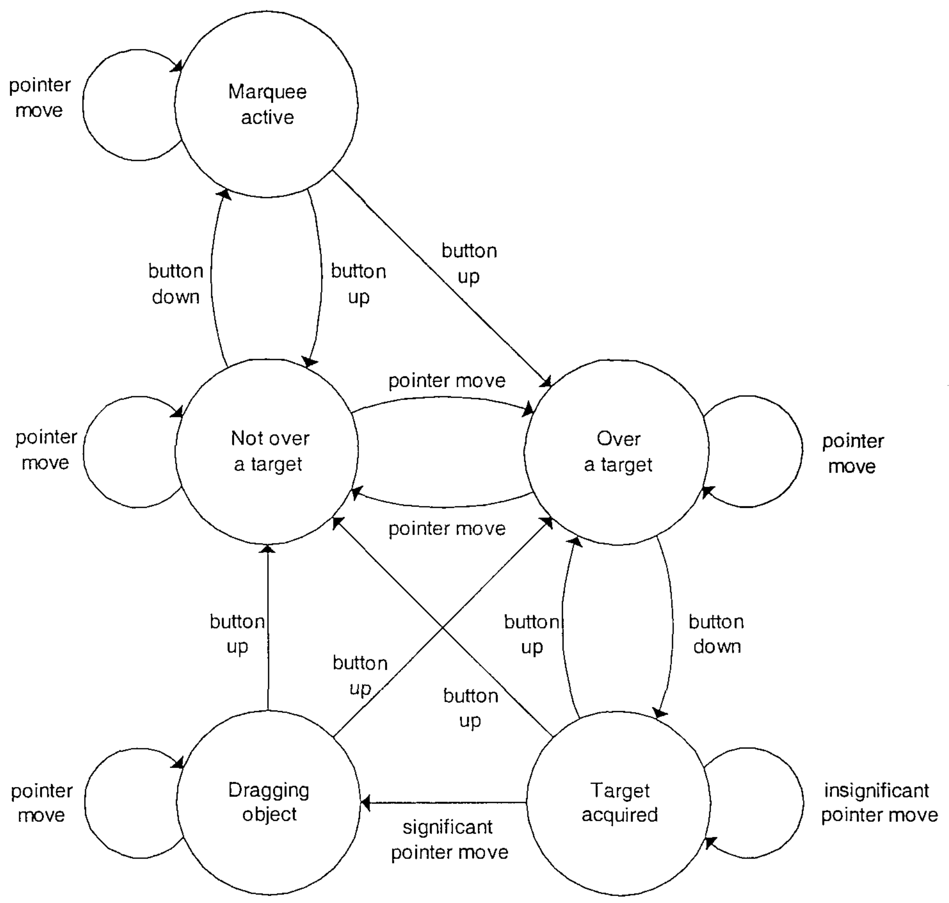


Figure 34 - State transition diagram for the Drag-and-Drop gesture

The origin of this behaviour might have been due to file integrity. Traditionally, the UNIX *mv* command allows moves only on the same mount point; because *mv* only modifies the *inode* structures to perform the move. The disk blocks containing the actual data of the file remain unmodified. Because of this, there is no actual change to the data and thus if the file contents were previously uncorrupted on the filesystem, then it would remain so. Efficiency in terms of speed may also apply.

The presence of this behaviour in the desktop environment is peculiar. The decision whether to move or copy might not be based on visual information. Two windows that represent two different folders might be visible, but they might not show any direct cues as to the volumes that they exist in. The default behaviour can usually be overridden by holding down specially designated keys to force either operation to occur, but this defeats the simplicity, if we were to qualify the gesture on every file system based Drag-and-Drop action.

Turning to the state changes in the input system during a successful Drag-and-Drop gesture, we note that the states for Drag-and-Drop are similar to Point-and-Click, except for a third state which is the "Dragging object" state. The difference in events is the addition of significant pointer movement in between the *button down* and *button up* states. There are two cases where a Drag-and-Drop gesture is not successful when the state toggles twice. Firstly, the user might not have been in the proximity of an object that could be dragged when the button was pressed. And secondly, the pointer might not have moved a significant amount of distance whilst the button was pressed. The second case is important, because the pointer movement during the button down state is what determines whether the gesture will be Point-and-Click or Drag-and-Drop.

Several sources [Cohen, Meyer, Nilsen 93] [Inkpen, Booth, Klawe 96] [MacKenzie, Sellen, Buxton 91] have stated that Drag-and-Drop is more likely to result in erroneous gestures over Point-and-Click. The cause of the problem is that the human hand is required to do two things during this gesture. Firstly, a digit is required to keep sufficient pressure on the device button whilst simultaneously moving the device to the desired location. The [Inkpen, Booth, Klawe 96] report has shown that stress levels are increased when the user is using the Drag-and-Drop gesture instead of the Point-and-Click gesture. This is apparent with a typical computer mouse. If the mouse button is sometimes defective, then additional mental stress is caused by the possibility of dropping an icon where we cannot easily recover from damage that would be caused by an erroneous drop.

The Drag-and-Drop gesture is equivalent to the gesture used to select items from menus. The user presses the button on a menu item and the menu opens up to reveal items that can be selected. When the button is released, the item under the pointer is selected. If the menu selection process is to be cancelled, then the pointer is moved away from the menu items and is then released. The actions that the hand must perform during this operation are precisely the same as the Drag-and-Drop actions.

The following quotation, from the Apple Tech Info Library [AppleTechInfoLibrary] shows why Apple introduced sticky menus to MacOS: The users wanted it.

> *"Apple added the feature of sticky menus because many of our customers requested it. It improves the human factors of the Mac OS because you do not have to hold the mouse button down so much." - Apple Tech Info Library, article 30202*

One might consider why *sticky Drag-and-Drop* has not been similarly implemented. In short, the problem is that it would conflict with the Point-and-Click gesture. We are able to find objects in the desktop environment that can be either selected or dragged. Upon button down and up, we would not know which gesture the user wanted, and particularly, we would not know whether to show the icon's image with the mouse pointer when it moved away from the icon's current location.

Such a gesture would have also introduced a new concept. The pointer would be capable of holding onto something without the pressure of the human digit. And more importantly, the act of something appearing "in the hand" need not necessarily be caused by the user's sustained pressure on the device button.

7.3 Drop-and-Catch

We introduce a new mechanism for direct manipulation interfaces: catch and drop. The user's model of this mechanism is simple. With Drag-and-Drop, you perform an operation with the mouse that you drag an object from one place to another. The object is held in the user's "hand" only during the gesture (whilst the mouse button is held down). The catch and drop mechanism assumes that the user can hold an object during normal tracking (i.e. with the mouse button released).

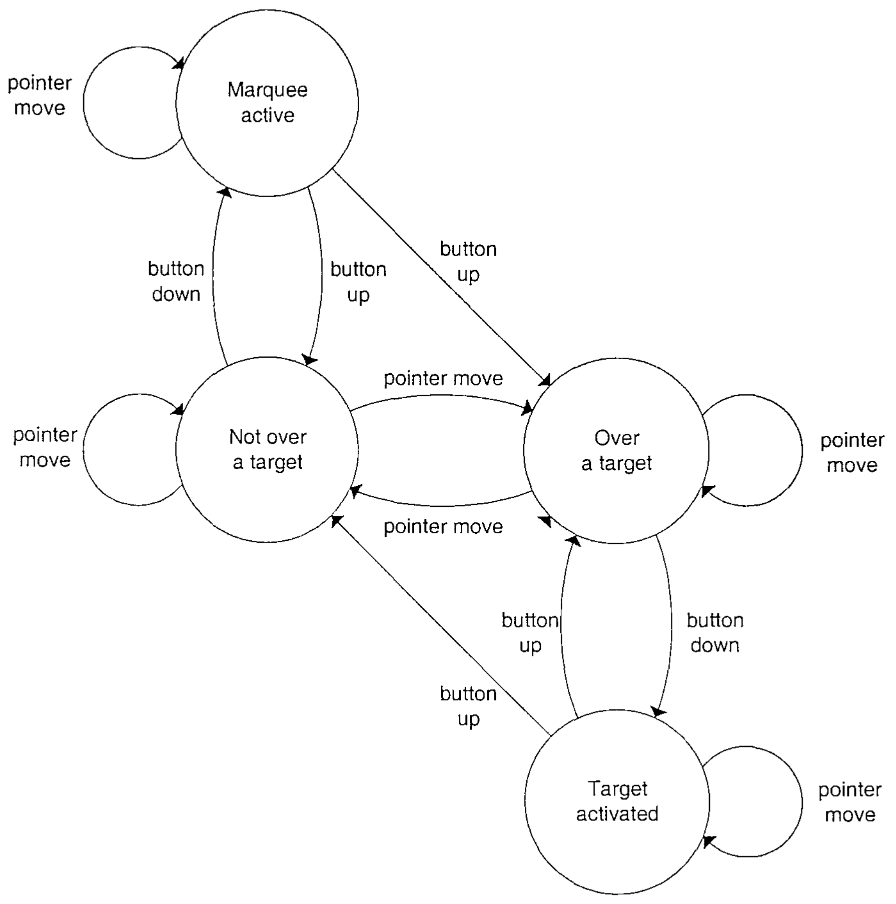


Figure 35 - State transition diagram for the Drop-and-Catch gesture

Drop-and-Catch is named such, because these are the two steps that the user's hand does when interacting with a desktop object. In the state transition diagram shown in Figure 35, the gesture is performed when the button is pressed over a target. The user drops the hand contents on an object by clicking on the object (button pressed and released). The object being interacted with can then throw the results back into the hand. When the mouse button is pressed onto an object, a request is sent to the object to respond. It can respond in three independent operations. Firstly, it can cause the hand to drop its contents (if held). Secondly, it can perform processing and thirdly, it can throw something back into the hand.

Drop-and-Catch is a gesture that is not used by today's desktops. The two major gestures involving the mouse are Point-and-Click and Drag-and-Drop. Selection is achieved by clicking on an object or by drawing a marquee around a number of objects. Once selected, the objects can be manipulated as a bag of objects. Whereas

Drag-and-Drop is achieved by pressing the mouse button on an object, or an existing selection, and holding the button down whilst moving the pointer to a drop destination.

There are three distinct stages during a successful *Drop-and-Catch* gesture:

(1) The user clicks on an object, and the contents of the hand is given to the clicked object.

(2) The clicked object has the opportunity to perform some processing, typically on the objects that have been "handed" to it.

(3) The user receives outputs from the clicked object, and they appear in hand. The results are then ready to be applied to a subsequent object.

In essence, if our hand is empty and we click on a desktop object, we are requesting for either the object to perform a task, to give us something in return for clicking it, or for both to occur. If we already have something in hand to give to the object, those objects are conceptually consumed by the desktop object as input material and for a subsequence side effect or further output to arrive in hand.

To facilitate moving objects between different displays, such as PDAs, the Pick-and-Drop gesture [Rekimoto 97] was proposed. This is essentially a sticky Drag-and-Drop gesture, to allow the stylus (in this case, a pen) to travel to another display during the gesture. One of the downfalls of Drag-and-Drop is dragging objects from one specialist application to another. Many of these programs can interact via Drag-and-Drop with the indigenous desktop environment, but we cannot be certain that two applications that have this behaviour will actually allow Drag-and-Drop between each other. In fact, Drag-and-Drop is a single gesture that involves two objects, the object being dragged and the object that it is dragged onto (in the case of a desktop window, we assume that the window is a container object). When the object being dragged is a virtual entity, it does not need to be rationalised until it is dropped.

7.3.1 Holding multiple objects

The Drop-and-Catch gesture as described above is similar to the command line pipe construct as found in a typical UNIX shell. Unfortunately, Drop-and-Catch also suffers a similar problem with shell pipelines in that it is difficult to describe processes that exhibit a non-linear pipeline. In a linear pipeline each component receives only a single object and generates a single object, and a pipeline can be constructed by simply clicking on each

As with the scripts in FCD Psh, there is no restriction for an object to receive and generate multiple objects, the limitation is due entirely to the human interface to the Psh environment. We do not see this as a limitation of the design of the Psh environment; we can at least define a possible mechanism to handle multiple objects.

We propose that the hand be actually a container that can hold zero or more objects. The hand contents is a sequence of objects. The sequence is important, because the sequence numbers will match the order in that icons were collected into the hand. A possible arrangement for applying processes, though we have not performed any research or implementation, would be to automatically collect multiple outputs into the hand as they are produced, and to give multiple inputs to directly manipulated desktop commands when possible.

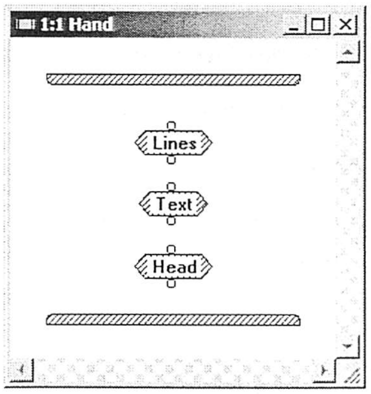


Figure 36 - Contents of the hand, shown as a folder of data items

Appending an object onto the end of the hand would be utilised by catching an object with a special qualifier held (like the *Control* key). This is a sequence that will normally invoke multiple selection on modern Windows. If the mouse contains more than one button, then we might also allow one of the buttons to "open up" the hand to show a linear sequence of iconic representations that can be manipulated as if it were a regular folder. Figure 36 shows what the hand might look like. In this case, we have collected a Lines document, a Text document and a Head command.

In a complete implementation of the Psh system (one that contains both the Prograph language, the Psh system and the direct manipulation interface), we envisage that content in hand could be modified directly in this form whilst creating a pipeline of processes. For example, running a Text document through the word counting command would produce three outputs: the number of lines, the number of words and the number of characters in the file. If we only require the number of characters for the next process, we could open the hand after the word count command has been used and directly remove the other two results from the hand. When we close the hand, the single result can then be used as input by itself for the next process.

7.4 History mechanism

An important concept in the UNIX shell is the history mechanism. When a command is invoked, it is stored within the command history of the shell. There are typically a number of ways to retrieve the history, for example the history command in which a user can reissue a command using a sequence number of where the command lies in the history. Some shells (for example, tcsh) have features to search for commands based on prefix search, and also to interactively browse the command history (for example, the bash shell).

In this section, we would like to consider the following scenario and how it interacts with the command history mechanism of the Psh environment. The user clicks on a Text document and drops it onto the *sort* command. The results are then dropped on the *head* command and then the results of that are dropped onto the *display* command. In this operation, the user has performed four pointer clicks.

These pointer clicks hold more information than those from the Drag-and-Drop gesture. Because intermediate objects are held in hand only between successive Drop-and-Catch gestures, this directly signifies the dataflow between the clicked command objects. When there is nothing left in the hand, then we know for sure that the current sequence has terminated. We know that the representations that are visually shown during a sequence of Drop-and-Catch gestures have a limited lifetime – up until the next gesture.

With this information, we can construct a script on the fly whilst the user is performing direct manipulation in the Psh environment. Once the current sequence of operations is finished, the script can then be placed into the command history buffer. The script resulting from the example above is show in Figure 37. Drag-and-Drop doesn't provide us with this information so accurately, as we would not know when the computation had finished.

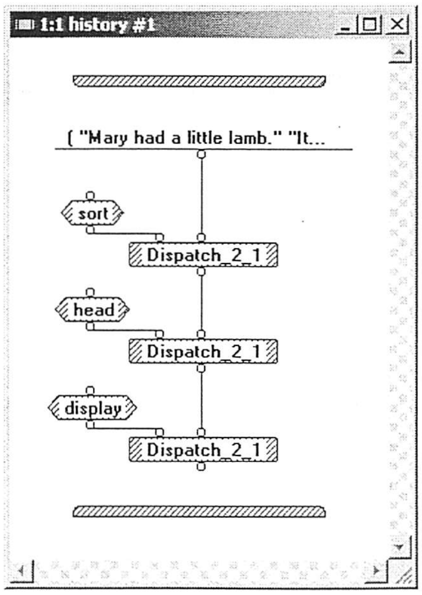


Figure 37 - Example script generated from direct manipulation

The command history is viewed similarly to the history in a UNIX shell, in chronological order. Since these are kept for all direct manipulation sequences in the Psh environment, users can copy a history item out of the command history and into their own workspace.

This is another piece in the puzzle for similarities between the UNIX shell and the desktop metaphor. The scripts held in the command history with the Psh environment are in the same form as those of the scripts that are built manually. We are able to manage a direct equivalent of the command history mechanism found in most UNIX shells, but all within the visual language of the desktop metaphor.

7.5 The Drop-and-Catch Demonstrator

To demonstrate the Drop-and-Catch gesture, we developed a demonstrator in Java [Sun]. The demonstrator was built using a Windows based system in conjunction with the Cygwin [Cygwin 03] environment.

The demonstrator mimics a typical desktop system, using common desktop utilities to view and edit files. The initial desktop state, shown in Figure 38, contains commonly found desktop utilities to delete objects (Recycle Bin), view objects (Explorer) and to edit objects (Notepad).

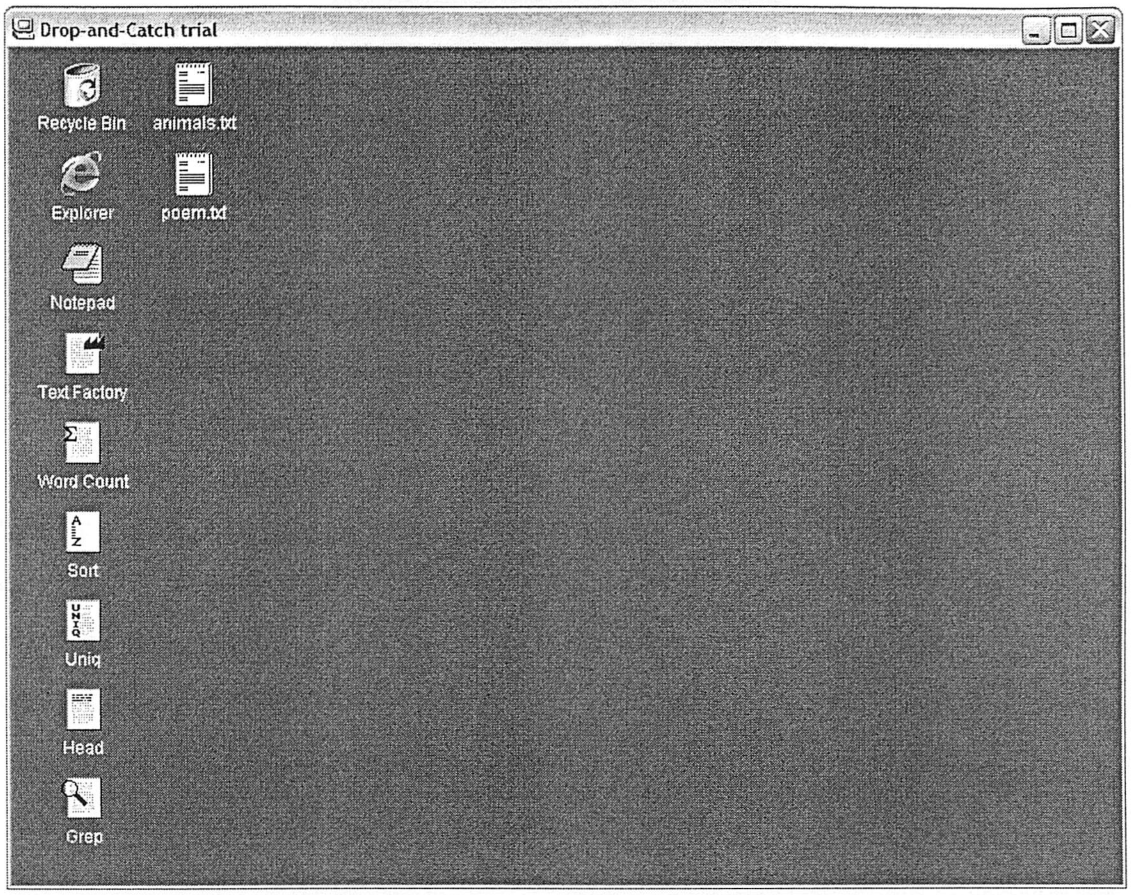


Figure 38 - The initial desktop configuration for the user trial

The demonstrator supports Word documents and Adobe Acrobat documents; though we could only view them with Internet Explorer as utilities to coerce those formats to text were not part of the Cygwin system at the time of writing.

Icons can be renamed by clicking on the label text, as shown in Figure 39. When objects are dropped onto persistent storage such as the desktop or a folder window, the label editor is invoked automatically with a name such as "unnamed01". This process forces the user to name objects on persistent storage.

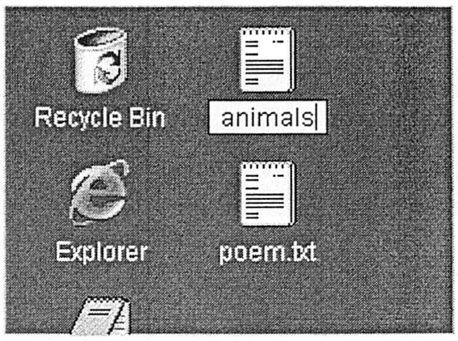
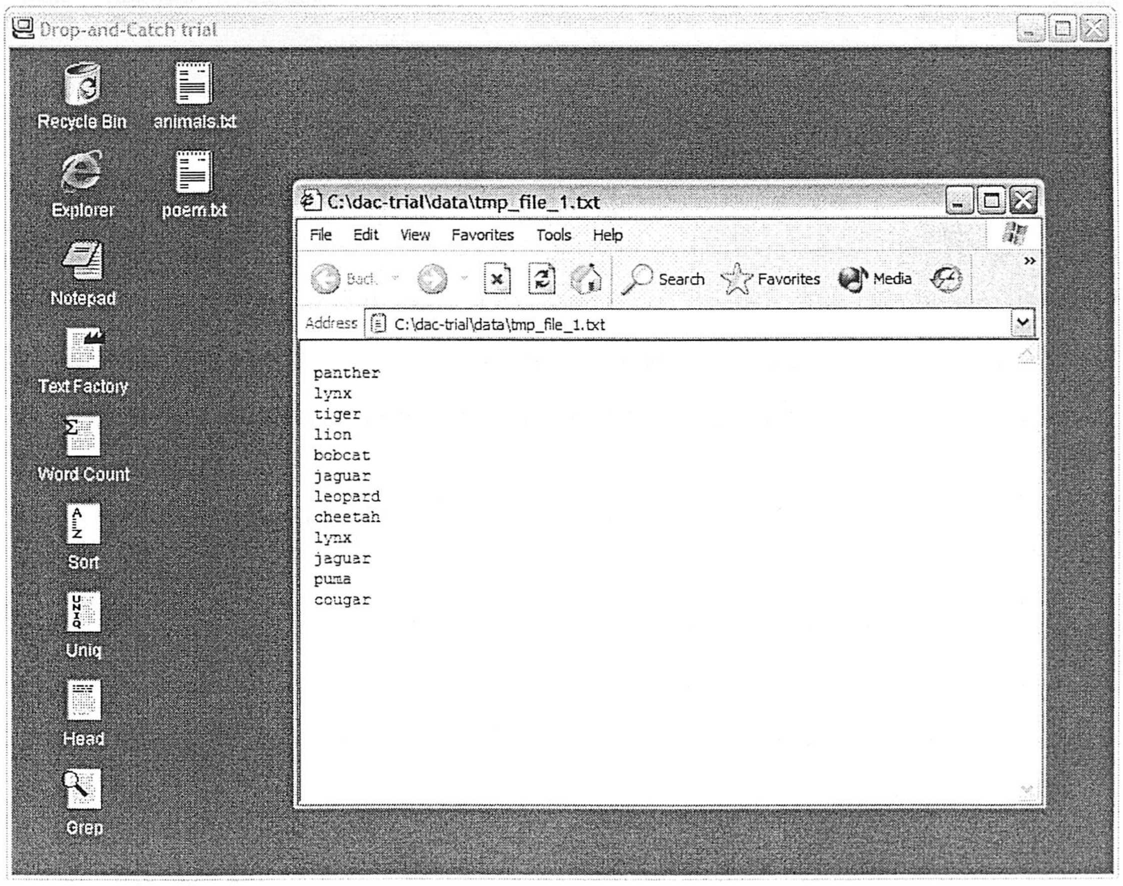


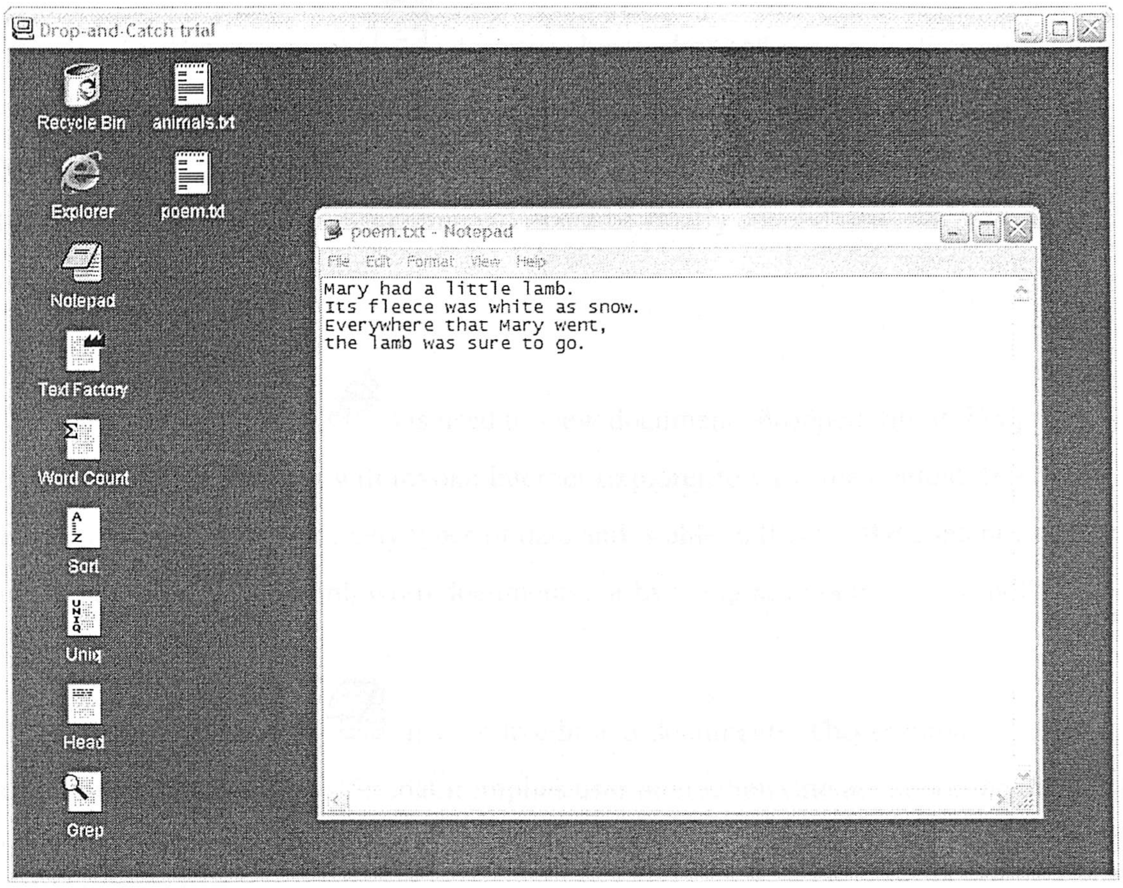
Figure 39 - Using the label editor to rename an icon

Objects can be viewed with Internet Explorer by using the **Explorer** command. An example of this is shown in Figure 40. Note that the URL shows a temporary filename. The reason for this is that a copy was taken of "animals.txt" by clicking on it and the copy was dropped into Explorer.


Drop-and-Catch trial
Recycle Bin
animals.txt
Explorer
poem.txt
Notepad
Text Factory
Word Count
Sort
Uniq
Head
Grep
C:\dac-trial\data\tmp_file_1.txt
File Edit View Favorites Tools Help
Back Search Favorites Media
Address C:\dac-trial\data\tmp_file_1.txt

```
panther
lynx
tiger
lion
bobcat
jaguar
leopard
cheetah
lynx
jaguar
puma
cougar
```

Figure 40 - Viewing objects with Internet Explorer



```
Mary had a little lamb.
Its fleece was white as snow.
Everywhere that Mary went,
the lamb was sure to go.
```

Figure 41 - Editing a text file with Notepad

Figure 41 shows a document editor with "poem.txt" opened. The filename visible in the Notepad window is "poem.txt", as the Notepad command was specially written to edit the originating document when a copy is dropped onto it.

7.5.1 Desktop commands

A number of desktop commands were implemented into the demonstrator. Not all commands were available in the user trial, as commands which were not required were hidden. Of the following commands, the Recycle Bin is implemented by using Java to delete files. Explorer and Notepad are utilised from the Windows environment and the remaining commands are drawn from the Cygwin environment.

The Recycle Bin command ( ) destroys objects dropped into it. Unlike the equivalent icon in the Windows desktop interface, objects are not stored in a separate location, but are destroyed immediately. In a full implementation, users may expect that objects dropped onto this command to not be finally deleted until the recycle bin is emptied.

The Explorer command ( ) is used to view documents dropped into it. Dropping objects onto the Explorer will invoke Internet Explorer to view the content. Internet Explorer is able to detect many types of data and is able to display the content using inbuilt code (e.g. text, html, word documents) or by using *plug in* code (e.g. pdf).

The Notepad command ( ) is used to edit text documents. This command can be differentiated from the rest in that it implies user interaction (the act of editing the text) in the midst of a Drop-and-Catch gesture. The consequences of this are discussed in Section 7.5.3.

The Text Factory command ( ) is used to create a blank text document. Clicking on this command causes the command to throw a blank text document into the hand. This is equivalent to selecting **New -> Text Document** in the contextual menu available in the Windows desktop and folder windows.

The Grep command (/) is used to select lines from a text document using a search pattern. This command requires two data items, a search pattern and a document to search within. For simplicity, the command was implemented with state so that two objects are dropped onto the command for it to operate. Firstly, the pattern is dropped onto the command in the form of a text file, and secondly, the document to be searched is dropped onto the command. Upon the second drop, the result is thrown into the hand. During the intermediate stage where a pattern is loaded into the command and it is awaiting the document to be searched, a **P** is shown on the icon.

The Gunzip command ( ) is used to unzip compressed documents. Compressed documents are decompressed using the Gnu GZip command and the decompressed copy is thrown into the user's hand.

The GZip command ( ) is used to compress documents using the Gnu GZip command.

The Head command ( ) command is used to truncate the given document to the first three lines. The number three was chosen as the default behaviour (UNIX uses 10 lines by default) for the purposes of the trial tasks. In a fuller implementation, a properties sheet would be used to change the number of lines to use.

The MD5 command ( ) is used to produce the message digest fingerprint (checksum) of a file, which is often used as a form of verification that the integrity of a file has been preserved during transfer across a network.

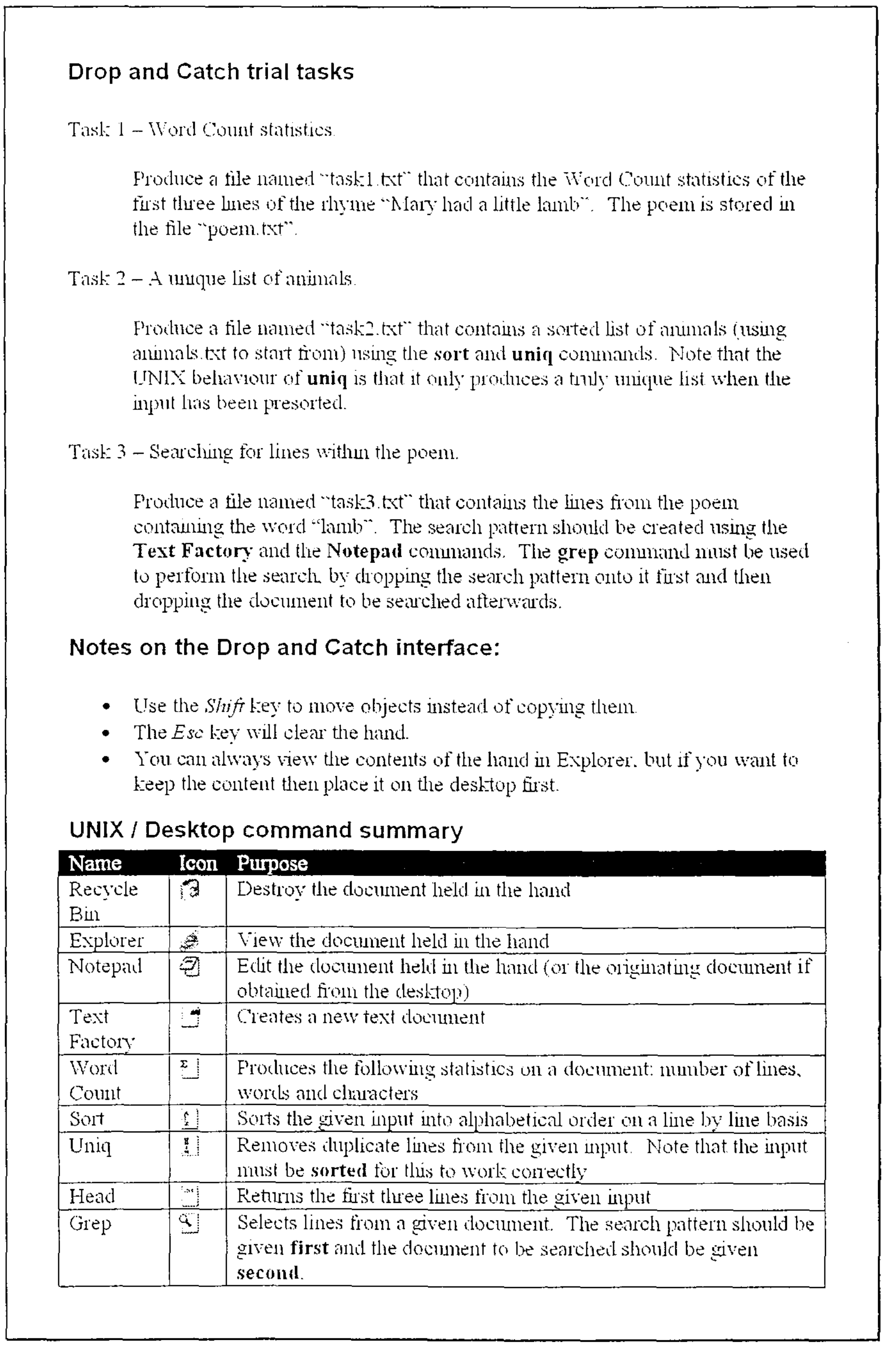
The Sort command ( ) is used to sort text files alphabetically on a per line basis. This command would be a candidate for properties such as reverse sorting available from a properties sheet.

The Uniq command ( ) is used to remove duplicate lines from a pre-sorted text file. The behaviour of requiring a pre-sorted text file is inherited from the UNIX command.

The Word Count command ( ) is used to perform word count statistics on the given document. It produces a text file containing a line of results. The results consist of the number of lines, the number of words and the number of characters in the given document.

7.5.2 User trial tasks

The tasks given to the trial user are shown in Figure 42. The user is given three simple tasks involving the desktop commands, simple interaction between data and these commands and storing the result of manipulating data with the desktop commands.



Drop and Catch trial tasks

Task 1 – Word Count statistics.

Produce a file named "task1.txt" that contains the Word Count statistics of the first three lines of the rhyme "Mary had a little lamb". The poem is stored in the file "poem.txt".

Task 2 – A unique list of animals.

Produce a file named "task2.txt" that contains a sorted list of animals (using animals.txt to start from) using the **sort** and **uniq** commands. Note that the UNIX behaviour of **uniq** is that it only produces a truly unique list when the input has been presorted.

Task 3 – Searching for lines within the poem.

Produce a file named "task3.txt" that contains the lines from the poem containing the word "lamb". The search pattern should be created using the **Text Factory** and the **Notepad** commands. The **grep** command must be used to perform the search, by dropping the search pattern onto it first and then dropping the document to be searched afterwards.

Notes on the Drop and Catch interface:

- Use the *Shift* key to move objects instead of copying them.
- The *Esc* key will clear the hand.
- You can always view the contents of the hand in Explorer, but if you want to keep the content then place it on the desktop first.

UNIX / Desktop command summary

Name	Icon	Purpose
Recycle Bin	🗑	Destroy the document held in the hand
Explorer	🔍	View the document held in the hand
Notepad	📝	Edit the document held in the hand (or the originating document if obtained from the desktop)
Text Factory	📄	Creates a new text document
Word Count	Σ	Produces the following statistics on a document: number of lines, words and characters
Sort	↕	Sorts the given input into alphabetical order on a line by line basis
Uniq	↕	Removes duplicate lines from the given input. Note that the input must be **sorted** for this to work correctly
Head	📄	Returns the first three lines from the given input
Grep	🔍	Selects lines from a given document. The search pattern should be given **first** and the document to be searched should be given **second**.

Figure 42 - Instruction sheet given to users before starting the trials

In the first task, the user is asked to produce word count statistics using the first three lines of the poem "Mary had a little lamb". The file is given on the desktop as "poem.txt", and the user is expected to click once on the **poem.txt** file to copy and catch it into the hand, drop and catch on the **Head** command to take the first three lines of the file, drop and catch on the **Word Count** command to produce the statistics, drop the result onto the desktop, and then name the result **taskl.txt**.

In the second task, the user is asked to produce a sorted list of animals with duplicates removed. The user is expected to click on **animal.txt**, drop and catch on **sort**, drop and catch on **uniq** and then to drop the result onto the desktop, naming it **task2.txt** in the process. Since we are using the Cygwin file utilities to perform these actions, our commands take on the behaviour of the UNIX utilities. The **uniq** utility requires the input to be sorted, thus our desktop command inherits the same behaviour.

In the third task, the user is asked to produce a file containing the lines of the previous poem which contain the word "lamb". This final task is more complex than the simple pipelines in the previous two tasks, it involves a desktop command that requires two separate inputs. The user is expected to create a new text file using the **Text Factory** command and place it onto the desktop (using a non specified name). Once it is on the desktop, they are expected to drop the file onto the Notepad command to change the file to contain the word "lamb". This behaviour is a departure from the Drop-and-Catch gesture as described previously in this Chapter, and will be discussed later in Section 7.5.3.

Once the file containing the word "lamb" is created, the user is expected to drop the file onto the **Grep** command, which adds a "P" character onto the iconic representation of the command to signify that it has a search pattern loaded. In this state, the user should then drop a copy of the **poem.txt** file onto **Grep** and then drop the resulting document onto the desktop and name it **task3.txt**.

7.5.3 External editing applications

The behaviour of Notepad is a departure from the Drop-and-Catch model as described previously in this Chapter. From a dataflow perspective, we cannot simply implement Drop-and-Catch on the Notepad command, because the processing between the drop and catch actions requires user intervention.

Notepad is special in that it tries to edit the document that originated the data rather than the data that is dropped onto it. The reason for this in the demonstrator is simply because Notepad is an external application and we don't know when the user has finished editing the file. The user might save the document and close Notepad – in which case we might check for changes in the modification timestamp of the file concerned. However, the user may save the file many times from the editor and we cannot predict this. Also we cannot rely on the user closing Notepad after they have finished with it, so monitoring the process list of the machine does not necessarily tell us when the intended edit is final.

The issue regarding Notepad concerns any external application that is used to edit files. Ideally the external applications would have their own method of letting the user catch data. BBEdit [Bare Bones] is able to drop text clips to the Macintosh Finder. Any text selection can be dragged to the desktop to create a text clip object in the Finder environment. A similar approach could be used in applications to throw data into the hand.

7.5.4 User trial results

Ten people were asked to perform the trial tasks. The tasks were trivial and all ten participants successfully completed the three tasks. Five of the users adjusted very easily to the demonstrator (showed quick understanding of the system before attempting the first task), two of the users adjusted easily (adjusted to the system by the start of the second task) and three of the users adjusted to the metaphor by the end of the second task.

7.5.4.1 Information shown during tracking

Four of the ten users commented that they found it hard to know when things had changed because the iconic representation hadn't changed. The "sense of state" isn't strong enough, according to one user. Another user stated that they wanted to click again because it wasn't obvious that any processing had taken place.

One user offered a possible solution to this problem: to have automatically generated icon names that represent the actions that the icon has been through. For example on Windows if you copy an object, e.g. "animals.txt", then the automatically generated name for the copied object is "Copy of animals.txt".

This would allow us to show an icon label during tracking, which would show the actions that the data has been through. In the first trial, the generated name for the result file might be

> "Word count statistics of the first three lines of poem.txt".

This might turn out to be too verbose for long pipelines, but it could serve as a line in a status bar area so that the user can check what they've done to the data so far.

7.5.4.2 Too easy to consume data during inspection

Three of the users found it too easy to consume data. Typically this occurred when they wanted to see what they had done to the data, and so went to view it using either Explorer or Notepad. One of the users suggested that inspecting the data should be a contextual menu option.

7.5.4.3 The special case for Notepad was difficult to guess

To four of the ten participants, the way to edit a file was confusing. Often I had to mention that Notepad was special (even though it is also written on the instruction sheet!) in that it modified the document on the desktop rather than the copy that is dropped into the Notepad command icon.

7.5.4.4 Visual cues to command state

One user commented that there are no visual cues as to how many actions are required on a command to make it “work”.

7.5.4.5 Double mouse clicks

Three of the users tried to double click on objects. This is most likely caused by habit in using double click in other desktop environments.

7.5.4.6 Real-time history viewing

One user would have liked to have seen a real-time history window building up, and for that history window to recall the data (as hover information) on the arcs.

7.5.4.7 Too easy to copy data

Five of the ten users found it too easy to copy data. The left mouse button in the demonstrator is set to copy data, and this was problematic. The user would put the icon down somewhere, and then quickly realise that they didn’t want it there. The immediate reaction was to click on the icon again to pick it up, thereby creating a copy of the data rather than moving it. Some users accidentally created up to 3 erroneous copies before forcing themselves not to instinctively click on the icon.

7.5.4.8 Fixed suffix during rename

One implementation detail that a user didn’t like was the fixed suffixes. The suffix (.txt in the trial) was fixed, but the icon label editor added it automatically after editing. The confusion was due to the fact that the extension was shown in normal mode but disappeared during edit mode. This was a decision I had made during the implementation of the demonstrator, as I did not want to add the possible confusion of data appearing as one type but having the data content of another type.

7.5.4.9 Copy / Move strategy

One user suggested that the demonstrator should have a more intelligent role in deciding whether to copy or move a document. Upon clicking on an object within a persistent container (e.g. the desktop or a folder window), the original object should grey out. The decision as whether to delete the original object or not depends on where the object is dropped. If the object is dropped on another persistent container, then the object should be moved. If the object is dropped onto a command, then the object should be copied. A special case of this is the Recycle Bin, which is designed to destroy the document placed into it. Without this special case, we could never destroy data once placed into a persistent container.

A strategy such as this has a number of benefits. Firstly the destiny of the original object isn't decided until it is dropped. Assuming that dropping an object onto a command will consume the object, then we note that the number of copies of a named object (e.g. an object that is contained in a persistent container) remains constant using the normal rules. Thus we don't experience the issue discussed in Section 7.5.4.7 where people erroneously made duplicate copies of data. Also we note that we cannot destroy documents held within persistent containers unless explicitly dropped onto the Recycle Bin. A further benefit is that data is retained if the data is forcibly dropped by the user emptying the hand (e.g. using the *Esc* key). In this case we assume that the user has made an error, or may have forgotten the task in hand (data can remain held in the hand when the user walks away). In the case that the document was taken from a persistent container, it would be returned.

7.5.4.10 When the Shift key is significant

One user informed me that the shift key was significant at the "start of the move" instead of the Windows behaviour where the move/copy/shortcut creation step isn't decided until the mouse button is released.

7.5.4.11 Didn't pipeline commands to start with

One user didn't pipeline the desktop commands to start with. The user placed intermediate results onto the desktop before moving the result into the next command. After the first trial, he realised that he could have done and then proceeded to pipeline commands for the second and third trial.

7.5.4.12 Hand tension

One user said that they suffered from the hand stress of dragging and was enthusiastic about the comparative lack of muscle tension required to perform the given tasks.

7.5.4.13 Visibility of temporary filenames

One user commented on the visibility of a temporary file name in the Word Count statistics. This was a small oversight in the implementation of the Word Count command. In the UNIX implementation of **wc**, the filename is shown if the input source was from a file, and nothing is shown if the content was taken from the standard input stream.

7.6 Implementation status

The visual test-harness to prototype the Drop-and-Catch gesture was implemented in C++, and communicates with the input device communications layer of the Amiga operating system. Later, a Java based implementation was written, and it is this version that is documented in this thesis and was used for the user trial.

7.7 Continuing Work for the demonstrator

The Drop-and-Catch demonstrator is ongoing work, and there are a number of issues and avenues to explore. In Section 7.5.4.1, we will implement a new function in the Command API which takes existing label names and produces new labels, for example the **Sort** command would turn “poem.txt” into “Sorted copy of poem.txt”.

This functionality, whether shown in a status bar or as a label with the icon during tracking, will strengthen the sense of state during operation.

The move/copy strategy discussed in Section 7.5.4.9 should eliminate the erroneous duplication problem discussed in Section 7.5.4.7. This will require the ability to show ghosted icons in the interface. As a consequence of this, the issue discussed in Section 7.5.4.10, where the shift key was significant at the beginning rather than the end of the tracking manoeuvre can be changed.

The addition of a contextual menu will allow us the option to view data during tracking and will help alleviate the problem of consuming data discussed in Section 7.5.4.2.

The issues discussed in Section 7.5.4.8 and Section 7.5.4.13 are small issues that shall be rectified.

7.8 Conclusions

In this Chapter, we have investigated the commonly used gestures of pointing devices in modern desktop environments. We have focused on the two main gestures, Point-and-Click and Drag-and-Drop. In relation to these gestures we have discussed the state transitions that occur in the input subsystem of an operating system when more than one gesture can be invoked at any one time.

Our system is a superset of the functionality provided by modern operating systems. The addition of the Drop-and-Catch gesture does not conflict with the existing gestures for marquee selection, Point-and-Click and Drag-and-Drop.

On the matter of the Drag-and-Drop gesture, we note the two simultaneous activities that the hand undergoes when performing this gesture with a mouse, and that experiments have shown that stress levels in users are higher with this gesture than equivalent Point-and-Click operations.

We found that users of the Apple Macintosh interface requested that Apple implement sticky menu selection into the Finder interface, due to usability issues. We then consider an equivalent modification to the Drag-and-Drop gesture. During this analysis, we refine the gesture to allow desktop objects to proactively throw objects into the user's hand.

The new gesture, named Drop-and-Catch, closely resembles the model used to construct pipelines within a UNIX shell. This is commonly regarded as a powerful construct to the UNIX philosophy, and with this new gesture, it can be directly mimicked in a desktop environment. The introduction of Drop-and-Catch into the input system does not interfere with other gestures such as marquee drawing, Point-and-Click and Drag-and-Drop. Our work is a superset of the normal functionality of desktop systems.

We then turn to the issue of multiple inputs and outputs in pipeline elements. This is a feature that even most UNIX shells do not exhibit a clean conceptual representation for. We show that the graphical nature of the desktop environment allows us to change our scope from the current manipulation to the hand so that currently held objects can be suitably reordered.

We then discuss the ease with which the direct manipulations made to the desktop environment can be recorded into a history of interactions, and how this is directly equivalent to the command history mechanism found in typical UNIX shells.

A demonstrator was built and a usability trial was performed on ten users, performing simple tasks with the new Drop-and-Catch gesture. The result of the trial was met with useful commentary on the user interface, with every participant able to complete the tasks using the Drop-and-Catch gesture.

8. Related Work

8.1 Introduction

In this thesis, we have discussed a number of areas including visual programming languages, programmable desktop interfaces, utilisation of distributed services and human / computer interaction techniques. We relate our work to other works from the literature in three major sections. We consider the "visual" aspect of our work. This includes using visual programming languages other than Prograph for implementation, existing desktop interfaces that exhibit programmability in non-textual form. We then place our work into context with other distributed systems. By distributed systems, we refer to services that are not local rather than focusing on parallel execution aspects. Finally, we contrast our work in the human interaction model with other works involving the usage of mouse-like pointing devices.

8.2 Visual Dataflow Languages

Visual dataflow languages are those that depict algorithms visually with directed graphs. The arcs of these graphs show dataflow throughout the graph, and will typically describe a process. We have seen the Prograph language extensively throughout this thesis, and we now examine two other visual dataflow languages, one of which has been designed to augment the desktop metaphor.

8.2.1 IShell

The IShell [Borg 90] is a tool for building visual pipelines. The dataflow metaphor is used to depict pipelines that do not conform to a linear sequence of processes. It is a closer analogy to the UNIX pipeline than the Prograph Psh prototypes in that the elements of the visual pipeline are actually running in parallel. A example script in the shell, called an *IScript,* is shown in Figure 43. In the script shown, input data is sorted and sent to a given user profile.

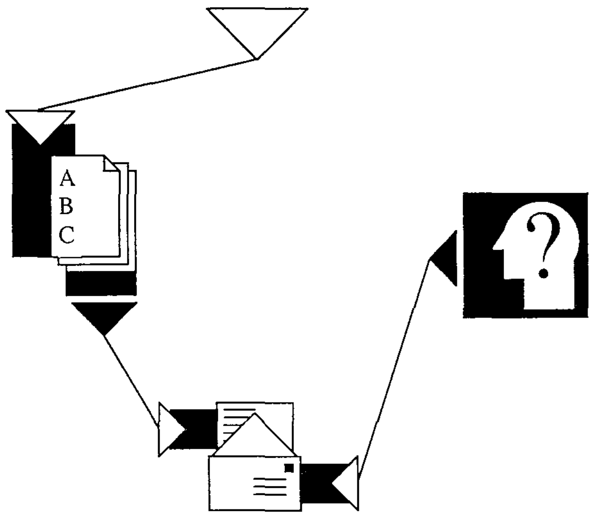


Figure 43 - An IShell script which sorts input data and emails it to a user

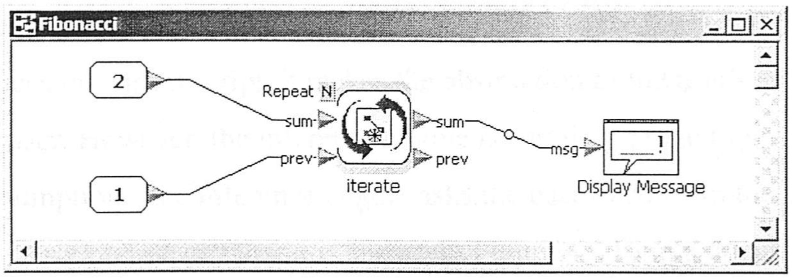
The IShell environment is focused on the concept of process control. Processes in the pipeline are executing in parallel, and the rate of flow between processes can be modified by the user to "fine-tune" the efficiency of the pipeline. Further to flow control, the user can click on an arc to view the actual of data flowing through the arc.

However, this close analogy with the UNIX pipeline is the cause of the IShell environment is incapable of coping with conditional algorithms. The IShell design does not mix data and control flow into a single graph, and thus it best describes a static network of ongoing processes. In line with its underlying host environment, Psh leans towards the pure visual dataflow model, and can be considered to be semantically equivalent to using only "pure" filters. A pure filter would be one that does not contain side effects other than output to the standard output stream.

The active entities in the IShell world are referred to as "desktop machines", and Borg cites the desktop printer as a simple desktop machine. A description of the sequence of gestures that the user might make to perform a task is included in this paper. The IScript language (the scripting language) is discussed, and mostly concentrates on fine-tuning the processes involved and does not discuss conditionals nor iterations.

8.2.2 Sanscript

We now turn to another visual programming language, but this time not in the context of a desktop replacement, but as a general purpose programming language. The Sanscript language is a visual dataflow language, and appears to be heavily influenced by the design of Prograph CPX. Sanscript shares many characteristics with Prograph, including the visual representations of the annotations that can be applied to the edges of nodes. An example, displaying a series of Fibonacci results is shown in Figure 44.



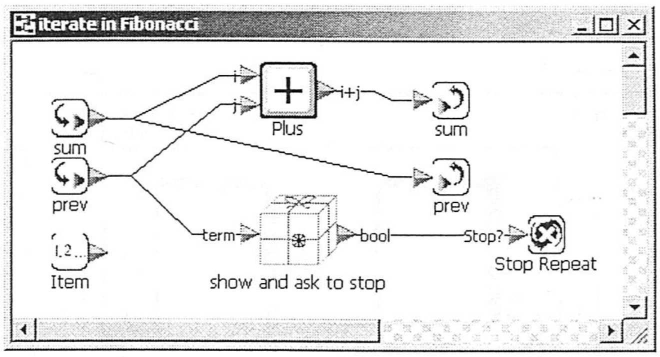


Figure 44 - Fibonacci in Sanscript

The major difference between Sanscript and Prograph is that Sanscript is a typed language. The type of each arc must be known, and fixed, for a Sanscript window to execute. If the data needs to be coerced, it must be done explicitly. In the Fibonacci example the little white circle between the *iterate* local and the *Display Message* method converts an integer to a string. Later revisions of the Prograph CPX language include a type tracing mechanism from the interactive editor, but these are only for information purposes.

Where types are coerced in Sanscript graphs, a filled circle is placed in the midpoint of the arc where the coercion occurs. The coercion methods are globally defined with no hierarchical patterns, and each coercion pair must to explicitly defined.

8.3 Non-Visual Dataflow Languages

Pursuit [Modugno, Corbett, Myers 95] is a "program by demonstration" shell that uses the comic strip metaphor. Scripts are built by recording the user's actions in the desktop environment and displaying them as a sequence of processes. To specify different cases, the user demonstrates the different cases to the PBD system. When Pursuit produces the visual script, it makes the abstraction of the user's actions on the behalf of the user. However, the inference engine is certainly prone to error. To avoid the wrong assumptions, the inference engine asks the user to confirm the abstractions that it makes. An example of Pursuit is shown in Figure 45. Here a document is duplicated, compressed and then stored into a backups folder.

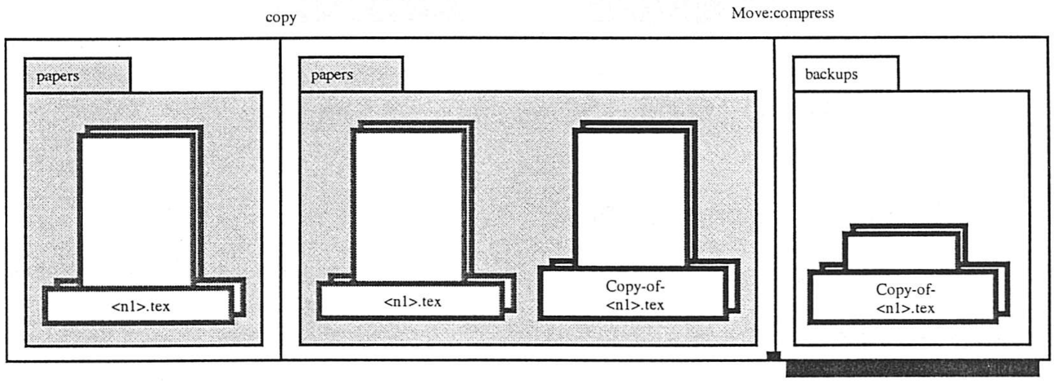


Figure 45 - Pursuit - compressing and making a backup

Pursuit addresses the problem of the complexity of the UNIX shell and how novice users find them cumbersome. It describes a UNIX based graphical shell replacement which records the user's actions and attempts to produce abstractions of them to turn them into scripts. Pursuit tries to bridge the gap between the power of UNIX and the conceptual simplicity of visual manipulation. The paper shows a graphical representation of the recorded script by using the comic strip metaphor. It claims an advantage of the system in that the visual scripts are more similar to the desktop environment than a textual shell. Each time an abstraction is made, a dialogue box appears so that the user can confirm or deny the proposed abstraction. The editor allows the user to explicitly create parts of a visual script so that cases which can not be inferred by the PBD system can be catered for. To reduce the visual clutter, data which is referred to several times (but is the same data) is the same colour. Data sets which are different are, of course, different colours.

SILK/JDI [Landay 96] also uses a PBD inferencing technique, but instead of recording operations from a desktop environment, the gestures are formed from raw pen strokes from an input stylus. Whereas Modugno's work concentrated on visual scripting, this concentrates on visual prototypes of graphical user interfaces. Hands strokes are directly interpreted as the description of a user interface.

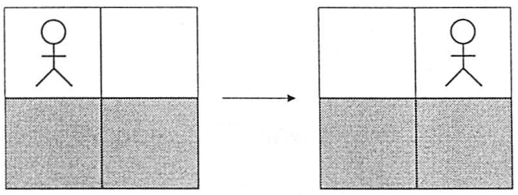


Figure 46 - A visual rewrite rule for COCOA

Rewrite languages are those that essentially transform a structure of one form to another by using a set of *rewriting rules*. If the structure is depicted visually, then we can consider such a language to be a visual language. The kinds of structures that can be transformed by rewriting languages depend on the language itself. COCOA [Canfield, Cypher, Spohrer 94] is an excellent example of a grid rewriting language, and an example "rule" is shown in Figure 46. The state of the language is represented by a two dimensional grid, and each cell contains some kind of state. Rules are defined to transform cells from one state to another depending on matched states and states of neighbouring cells. In this case, the stick figure will move right if there is a free space to the right of the figure, and there is ground underneath both the stick figure's current position and the destination. PROGRES [Schürr 97] is a combined visual graph rewriting language and a textual language. The graph rewriting component allows for fully general graphs. The Aardappel [Oortmerssen] language works on tree structures, and the rules can contain special annotations for concurrent rewriting of tree structures.

8.4 Direct Manipulation

Turning to the direct manipulation of iconic environments, we examine two gestures that have been suggested for use in the IShell world and also for portable digital assistants.

The IShell design extends to a consideration of how the icons in the environment are manipulated. In essence, the gesture described is the Drag-and-Drop gesture. Compared to our Drop-and-Catch gesture, we can envisage that icons that represent processed output would simply fall onto the desktop close to the desktop machine that generated them.

The Pick-and-Drop gesture [Rekimoto 97] amounts to a "sticky" Drag-and-Drop gesture. The purpose of the sticky element is so that it can be implemented on PDAs so that a pen can select an icon on one PDA and drop onto another. This is a preferable way to perform file transfer between PDA devices, as it closely matches how a file would be copied or moved locally within the device. In comparison with the Drop-and-Catch gesture, we note that desktop objects cannot "throw" objects onto the pen.

8.5 Conclusions

In this chapter, we have examined a variety of related technologies. We have examined the major types of visual programming languages: graph flow, sequential and rewrite languages. We examined a graph flow desktop interface, that is similar to the Classic Psh design, and a visual dataflow programming language in a similar vein to the Prograph language. We noted that the comic strip metaphor is a special case of the graph flow language, and we examined a desktop macro recorder and a user interface builder using the comic strip metaphor. We discussed the merit of the comic strip metaphor in that it can encapsulate a linear sequence of actions given by the user and present the sequence without layout issues. To conclude our examine of visual programming languages, we looked at graphical rewrite languages that perform transformation on visual structures.

We then turned to the direct manipulation of desktop interfaces, thereby examining the gestures that are used to program a graph flow desktop interface and for portable digital assistants. We noted that both systems essentially use the Drag-and-Drop gesture, with the latter introducing a "sticky" element to allow objects to be transferred between PDAs.

Finally, we looked at attempts to classify the services provided by distributed systems such as the internet and how the Psh environment maps onto such a structure.

9. Continuing Work

In addition to the continuing work on the Drop-and-Catch demonstrator (discussed in Section 7.7), there are a number of future directions that we are considering for this research:

9.1 Password persistence

As Psh allows the user to create scripts that do not explicitly state an access scheme, it is feasible that the user might use such a script on files via authenticated FTP access. There is a conflict between user convenience and password security. It is unreasonable to expect the user to enter a password for every operation performed on the FTP site, but the user also needs to know when the password has been "forgotten" by the system.

9.2 Markup languages

Another area for investigation is markup languages. In Chapter 4, we developed an HTML suite for Psh for scripts to render HTML. By generalising the current suite, we could create a general document mark-up suite and represent document elements within the data class hierarchy of the later variants of Psh.

There are two roles for such an extension: firstly we need to be able to generate documents in various mark-up languages, like RTF and *latex*. Secondly, we want to be able to handle existing documents in these mark-up languages. We do not want to create a canonical document mark-up language, and in some cases, it may not be possible. For example, we might not be able to describe all types of tables from latex in the language of HTML. The implementation would utilise the lazy coercion offered by DDD Psh and FCD Psh to leave document elements defined in their own document language as much as possible.

9.3 Distributed / remote execution

Psh has already moved the desktop environment closer to the internet/intranet by the use of URLs as the general resource identifier. The desktop printer is already a kind of intranet based desktop object. We can create print jobs by dragging documents onto the printer icon and monitor their status by opening the icon. This is an example of interaction with a remote service directly from the desktop environment. Work needs to be done on the usefulness of distributing processes in general at the desktop environment level. One possible implementation of this might be to make some desktop commands accessible by a new URL scheme, say `command:`. Using a separate URL scheme would improve the recognition of remote commands over the current *cgi-bin* implementation.

On the issue of distributed flow of data between desktop, we consider that streaming data across the internet is a developing area of research, and using the Psh environment as a composition tool to prototype such networks would need further research. It is known that Pictorius are extending the current failure mechanism in the form of *exceptions*. Prograph is currently limited to a linear sequence of case windows, and that more than one kind of failure condition can only be modelled by using the global state to carry such information.

9.4 Identify a useful set of desktop commands

A continuation of this project will be to identify a useful set of desktop commands. For general text based operations, we can look back in hindsight at the successful utilities from UNIX. However, with datatypes such as pictures and sounds, we would need to reconsider how many of those utilities would operate with diverse datatypes. For example, the word count utility would use techniques similar to Optical Character Recognition for pictures, and speech interpretation for sounds.

One consideration is that the Psh environment may be composed of several components from different vendors, so some parts of the command hierarchy might need to be managed by a neutral organisation.

9.5 Deployment in alternative environments

We have focused on the desktop environment as a primary target for Psh, as it was where we found the need most pressing. Other environments targeted could be development environments. An investigation into the language of Make would be an ideal candidate for this in conjunction with an investigation into the extent of side-effects.

9.6 Distributed Systems

We have described Psh as an enhancement to the desktop metaphor. Challenges for global computation are different. Latency and bandwidth are limiting factors, instead of processor speed and memory. Cardelli summarises the challenges as follows: (1) Typing. It states that MIME is used throughout the internet, but most web servers poorly construct the MIME information for documents (true). (2) Security. "paranoia rules". There are no global security models that cover the network protocols to the application layer. (3) Reliability. (4) Modularity. By this, he means program code being retrieved when necessary. And (5) Resource Management. Time, space, bandwidth and services need to be managed. These issues highlight that the Web is a distributed system where each resource can fail or provide under par service.

9.7 Extent of side-effects

Prograph is an unpure dataflow language. The syntax of Prograph can indeed be interpreted as pure, but the implementers decided on mutable objects and shallow lists. An investigation into this would provide useful information on the Prograph development environment.

9.8 Implementation

We are considering the option of integrating FCD Psh with the Macintosh Finder, by using Apple Events, to provide a visual alternative to the AppleScript language.

10. Conclusions

In this thesis, we have examined the visual dataflow metaphor as a means of enhancing the usefulness of the desktop metaphor. We introduced the concepts and the usage of scripting languages in current desktop environments, such as the UNIX shell, the MS-DOS shell and the AppleScript language. We noted the impact of the URL in unifying the access remote services into a convenient string based descriptor, and the underlying internet protocols.

There is a significant overlap between the features of the desktop environment and the visual language paradigm; we have noted the similar paradigm of using iconic representations to depict a related set of documents or an executable program respectively. We contrast the dataflow and the comic strip metaphor and give reasons for our preference for the former as the visual metaphor for the following work.

In the Prograph chapter, we gave an introduction to the visual programming language of Prograph. Prograph is a object oriented visual dataflow language, and we discussed the run-time creation and modification of classes in the Prograph environment. Of the language features, we focused on the Data Directed Dispatching mechanism, and how it obtains typeless parametric polymorphism.

After the introduction to the visual dataflow language, we turned to the existing Psh prototype, that we subsequently named Classic Psh. Programmability already existed in the world of the Finder, but only in the textual form of AppleScript. The Psh system extended the programmability of the Finder whilst remaining in the visual domain. Classic Psh provided a single cognitive model combining the desktop metaphor with scripting ability. We noted that there were some novel features of Psh over typical UNIX shells, such as multiple return values and cleaner syntax for reusing output from computations as a result of the visual representation. We then discussed the document generation component that generates hypertext documents. The HTML suite can be viewed as both a collection of very simple functions, but with Prograph annotations, a powerful suite to express the structure of documents that exhibit regular structure.

In the next chapter, we discussed the implementation of DDD Psh, an object oriented iteration of the Psh environment that utilises the semantics given by the Prograph OO mechanism. In addition to the use of the class hierarchy to describe kinds of data, we described a convention for coercing objects of one class to another. To demonstrate the design, we implemented a number of scripts for DDD Psh and discussed the issues that arose during the process. We noted the issue of independent authorship of functional packages, primarily requiring the need to modify existing classes in some circumstances.

To address these issues, we designed and implemented FCD Psh. In Chapter 6, we discussed the requirements that were not met with the DDD Psh design, and identified some scripts that exemplified the problems with DDD Psh which were solved by FCD Psh.

The FCD Psh environment can be enriched with functionality towards a specific problem domain by adding a Prograph section. The Prograph 'section' is a useful mechanism, as it keeps all of the classes and associated methods together in a single module that can be installed or removed simply by adding or removing the section from the Psh environment.

In Chapter 5, we first described refinement of the object oriented Psh environment. The initial OO model, DDD Psh, was implemented using the inbuilt OO mechanism supplied by the Prograph language, and was characterised by the data directed dispatching mechanism. During the course of developing DDD Psh, we needed to modify existing classes to allow two way coercion between the existing and newly developed classes. This became apparent when we added the new functionality via Prograph sections. Our efforts to redefine the existing classes to allow this kind of reuse were unsuccessful, and so the design of FCD Psh focused on a Psh environment that would correctly support the scripts. The classes were authored using our objectives of non-modifiable and mutually coercible class structures. To achieve this, we designed and implemented a new OO methodology. The most prominent feature of this new OO mechanism is the first class dispatcher. The new OO mechanism retains many important properties of object orientation, and can even support interoperation of different kinds of object orientation methodologies.

We investigated the commonly used gestures for pointing devices, in both state behaviour and physical stress induced by the user in using these gestures. We proposed a new gesture *Drop-and-Catch* to keep embodiments of data in the pointer device between successive command invocations. Extensions to the model are suggested, with multiple items in the hand and also a history mechanism where scripts are stored and process visually. We implement a demonstrator for the Drop-and-Catch gesture and conduct a usability trial. Positive commentary is received and a number of future directions are proposed for this research.

DDD Psh and FCD Psh were implemented during the course of this research within the visual language of Prograph. We propose and describe the implementation of a clean separation between access type and content type. We are able to write visual scripts that can process data on the internet/intranet, provided the protocol used supports our actions. In cases where such actions are not permitted by protocol, we are able to express alternative behaviours. We have indicated how the use of data-directed dispatching offered by object-oriented facilities can be used to generalise and simplify the mapping of data to applications.

11. References

[Ambler 97]	Ambler, A.L. et al. *1997 Visual Programming Language Challenge*. IEEE. 1997.
[Amiga]	Amiga Inc. URL "http://www.amiga.com/"
[Appleevents 94]	Apple Computer Inc. *AppleScripting and Apple Events*. 1994.
[AppleTechInfoLibrary]	AppleCare Knowledge Base. Document 30202. URL "http://docs.info.apple.com/". 1997.
[Bare Bones]	Bare Bones Software, Inc. http://www.barebones.com/
[Berners-Lee, Fielding, Frystyk 96]	Berners-Lee, T., Fielding, R. and Frystyk, H. *Hypertext Transfer Protocol*. RFC 1945. URL "http://web.mit.edu/rfc/rfc1945.txt". 1996.
[Berners-Lee, Masinter, McCahill 94]	Berners-Lee, T., Masinter, L. and McCahill, M. *Uniform Resource Locators*. RFC 1738. URL "http://web.mit.edu/rfc/rfc1738.txt". 1994.
[Borg 90]	ISHELL Borg, K. *IShell: A Visual UNIX Shell*. In Proceedings of ACM CHI'90 Conference on Human Factors in Computing Systems. 1990.
[Canfield, Cypher, Spohrer 94]	Canfield, D.C., Cypher, A., Spohrer, J. *KidSim: programming agents without a programming language*. In Communications of the ACM. Vol. 37, No. 7, Pages 54-67. 1994.

[Cardelli 97a] Cardelli, L. *Global Computation.* In ACM SIGPLAN Notices. Pages 66-68. 1997.

[Cohen, Meyer, Nilsen 93] Cohen, O., Meyer, S., Nilsen, E. *Studying the movement of high-tech Rodentia: pointing and dragging.* In Human Factors in Computing Systems (CHI 93). Pages 135-136. 1993.

[Cruickshank, Glaser 99] Cruickshank, D., Glaser, H., *Direct Manipulation, Scalability and the Internet.* In Lecture Notes in Artificial Intelligence, No. 1624. Pages 102-112. 1999.

[Cygwin 03] http://www.cygwin.com/

[Fertig, Freeman, Gelernter 96] Fertig, S., Freeman, E., Gelernter, D. *Lifestreams: An alternative to the Desktop Metaphor.* In Proceedings of ACM CHI 96 Conference on Human Factors in Computing Systems. Pages 410-411. 1996.

[Fielding 95] Fielding, R. *Relative Uniform Resource Locators.* RFC 1808. URL "http://web.mit.edu/rfc/rfc1808.txt". 1995.

[Foley 90] Foley, J. D., van Dam, A., Feiner, S. K., Hughes, J. F. *Computer Graphics, Principles and Practice.* ISBN 0-201-12110-7. Second edition. Pages 564-568. 1990.

[Fox, Furmanski 96] Fox, G. and Furmanski, W. *Towards Web/Java based High Performance Distributed Computing - an Evolving Machine*. In Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing. 1996.

[Freed, Borenstein 96] Freed, N. and Borenstein, N. *Multipurpose Internet Mail Extensions*. RFC 2045. URL "http://web.mit.edu/rfc/rfc2045.txt". 1996.

[Gentner, Nielson 96] Gentner, D. and Nielson, J. *The Anti-Mac Interface*. In Communications of the ACM, Vol. 9, No. 8, Pages 70-82. 1996.

[Giles, Cox, Pietrzykowski 89] Giles, F., Cox, P., Pietrzykowski, T., *Prograph: A step towards liberating programming from textual conditioning*. In Proceedings of IEEE Workshop on Visual Languages. Pages 150-156. 1989.

[Inkpen, Booth, Klawe 96] Inkpen, K., Booth, K., Klawe, M., *Drag-and-Drop vs. Point-and-Click Mouse Interaction for Children*. Technical Report TR-96-20. Department of Computer Science, University of British Columbia. 1996.

[Korn, Northrup, Korn 96] Korn, D. G., Northrup C. J. and Korn J. *The New* Korn *Shell*. In Linux Journal volume 27, July 1996.

[Lee, Morris 99] Lee, G., Morris, J., *Dataflow Java: Implicitly Parallel Java*. Technical Report TR99-01. Centre for Intelligent Information Processing Systems, University of Western Australia. 1999.

[MacKenzie, Sellen, Buxton 91]	I. Scott MacKenzie, Abigail Sellen, William A. S. Buxton. *A comparison of input devices in element pointing and dragging tasks*. In CHI '91. Human factors in computing systems conference proceedings on Reaching through technology, pages 161-166. 1991.
[de Mey, Nierstrasz 93]	de Mey, V., Nierstrasz, O. *The* ITHACA *Application Development Environment*, Centre Universitaire d'Informatique, University of Geneva, Technical Report page 267-280, July 1993.
[Modugno, Corbett, Myers 95]	Modugno, F., Corbett, A.T. and Myers, B.A. *Evaluating program representation in a demonstrational visual shell*. In Human Factors in Computing Systems (CHI) - Conference Proceedings. 1995.
[Oortmerssen]	van Oortmerssen, W. The Aardappel Programming Language – Concurrent Tree Space Transformation. http://wouter.fov120.com/aardappel/
[Prograph 93]	Prograph International. *Prograph CPX User's Guide*. 1993.
[Rekimoto 97]	Rekimoto, J. *Pick-and-drop: a direct manipulation technique for multiple computer environments*. In UIST '97. Proceedings of the 10th annual ACM symposium on User interface software and technology, pages 31-39. 1997.

[Schmucker 96]	Schmucker, K. J., *Rapid Prototyping using Visual Programming Tools.* In Proceedings of the CHI '96 conference companion on Human factors in computing Systems. 1996.
[Schürr 97]	Schürr, A. *Developing graphical (software engineering) tools with PROGRES.* In Proceedings of the 1997 international conference on Software engineering.
[Shapiro 95]	Shapiro, J. *Prototyping Algorithms in Perl.* In Linux Journal. Aug 1995.
[Shivers 94]	Shivers, O. *A Scheme Shell.* MIT Laboratory for Computer Science Technical Report 635. 1994.
[Stairways 99]	Stairways Software Pty Ltd. *Anarchie 3.7.* http://www.stairways.com
[Steele 90]	Steele, G. L. *Common Lisp the Language.* Digital Press. ISBN 1-55558-041-6. 1990.
[Sun]	Sun Microsystems, Inc. http://java.sun.com/
[Sussman, Steele, Gabriel 93]	Sussman, G. J., Steele Jr., G., L., Gabriel, R. P., *A Brief Introduction to Lisp.* In the 2nd ACM SIGPLAN History of Programming Languages Conference. ACM SIG-PLAN Notices. Volume 28. 1993.

[Wagner, Curran, O'Brien 95] Wagner, A., Curran, P., O'Brien, R. *Drag Me, Drop Me, Treat Me Like an Object.* In Proceedings of ACM CHI'95 Conference on Human Factors in Computing Systems. Pages 525-530. 1995.