Generalising a File Manager into an Address Book and Other Things

J. D. Newmarch
Faculty of Information Science and Engineering
University of Canberra
PO Box 1 Belconnen
ACT 2616 Australia
email: jan@pandonia.canberra.edu.au

Introduction

Last year, I designed and implemented a file manager called `xmfm`[1] (X/Motif File Manager – see Figure 1). This was released on alt.sources, and has received a favourable reception. It has been in use by staff and students at the University of Canberra for some time. The next section describes `xmfm` and other file managers in brief.

After a considerable period of use, I have been able to look at the good and bad features of `xmfm`. In general, the bad features are not significant but they they have had an effect on a certain class of user: the Unix "power users," who have extensive experience with the older command line interface, and who have adopted the newer shells which extend in subtle ways the older shells. For these users (myself included), `xmfm` is simply too monolithic and inflexible, and so these users continue to use a number of `xterm`'s running the Korn shell, the `tcsh`, `bash` or the Z–shell.

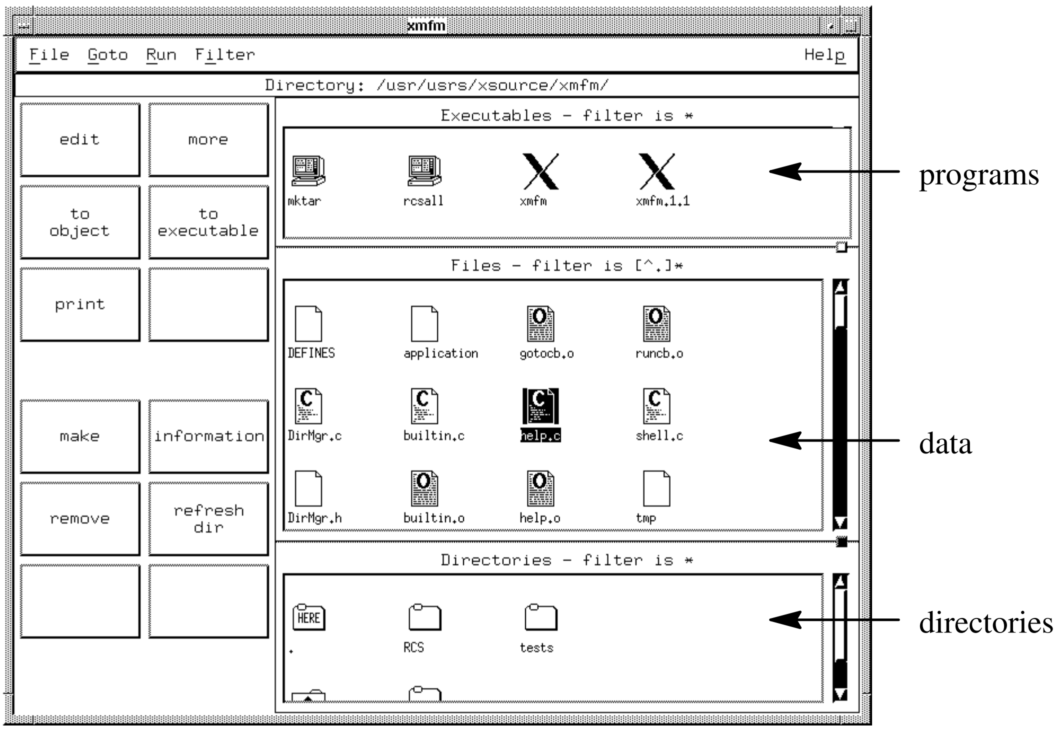

Figure 1: xmfm

Without discarding xmfm for the more casual user, this paper reports on a redesign that is intended to allow the best features of command line environments to be used and bring in graphical features where appropriate. It forms a component of a project XBatch which aims at bringing graphical and command line interfaces closer together.

The new design allows relevant bits of a file manager to be used where appropriate. In addition, the bits are highly configurable and may be used in quite different ways to the original design. The paper discusses some alternative uses such as an address book.

X File Managers

There are a number of file managers available for X nowadays. There is the filemgr from Sun for the OpenLook environment, Looking Glass from Visix, X.desktop from IXI, the freely available xdtm, and the freely available xmfm from the author.

These file managers all share common features: they give a graphical display of the files in a directory, and when a file is selected allow a "suitable" application to be started on or using the file. Beyond this, they begin to diverge.

xdtm, for example, adopts a "task oriented" model, in which the user has a set of tools applicable to a task that can be invoked on the selected files. In a "program building" task the tools are editors, compilers and debuggers, and they can be applied to any file in the display.

X.desktop and filemgr attach a single application to each file that can be invoked. xmfm and Looking Glass attach a set of applications to each file so that one of this set can be invoked on the file.

xmfm and Looking Glass allow direct access to a shell interpreter from a pull–down menu for commands that are not supported directly by the file manager. Looking Glass and X.desktop provide additional tools, and filemgr exists (usually) in the rich OpenLook environment.

Command Line Interpreters

The Unix shell interpreters have always been powerful, if a bit quirky. The current generation such as the Korn shell have added job control mechanisms, filename completion and easy history editing to the standard repertoire of loops, pipes and command evaluation. Next generation shells such as es[2] are under development.

Features such as command line editing allow repetitive tasks to be performed very easily. When the edit/compile/debug cycle is contained within two "up arrow" key motions it is not clear that the mouse "select and double–click" is really an improvement. In addition, random tasks can be accomplished very quickly within this model by just typing a different command. Finally, activities can be suspended and resumed using the job control mechanisms of these shells.

However, the command line interface is clearly not ideal. Job control is fairly crude and was beginning to give way to virtual consoles which could be switched between. Now one can just switch focus on xterm's. More seriously though, they are not good for display of data. For example, a common command–line cycle is `ls, vi, ..., ls`, where the repeated `ls` commands are due to the loss of information due to other activities. Even binding the PageUp key to the `xterm` scrollbar doesn't really help.

Problems with File managers

File managers have nowhere near the flexibility of the command line interface. The command line accessible from `xmfm` and Looking Glass requires extra keystrokes or mouse actions to reach (the fastest is `Meta-C` in `xmfm`). It would be possible to add keyboard accelerators such as "!!" to xmfm, but such changes are adhoc, and anyway already exist in most shells. The real problem is that the shells already do this well: why have to rebuild it afresh?

All file managers take up a large amount of real–estate in showing directories. The addition of menu–bars, etc uses even more space. `xmfm` uses a substantial amount in showing the possible actions that can be performed on a file. On a large high–resolution screen this is not a serious problem, but on smaller screens it is. This offends the "parsimony" principle that many Unix people work under. Using space when necessary is fine, but wasting space is not.

One Internet posting seemed to suggest that what the poster used in `xmfm` was firstly its graphical display of objects and secondly the inclusion of Roger Reynolds' Drag and Drop, which allowed `xmfm` to be used with other tools; the other facilities in `xmfm` were not mentioned – were they of any use to this user? If not, should they be there?

The Components of a File Manager

The best component of a command–line system is its flexibility. It will be hard to duplicate this in a windowing system. Therefore a "power user" will often still drive out of a command line. Where the command line loses advantages is in the display of data. A primary component of a file manager is this display activity.

But display of what? Display all files, or just a subset? All file managers have a filter mechanism, which duplicates to greater or lesser extent the shell pattern matchers. It would be far simpler to let the shell perform the match, and just use its output for display.

Thus, the primary component is as a graphical display of distinct pieces of information, supplied by some input source.

The information supplied is normally categorised: for example, a different icon for C source code files to the icon for tar archive files. The mechanisms to perform this categorisation differ widely: `xmfm` uses shell file patterns whereas Looking Glass and `X.desktop` use proprietary command languages that can peek into files as well as perform simple pattern matches. Whatever mechanism is used, the result is also a feature of file managers: they classify input data into classes, with a distinct icon for each class.

Thus, the second component of a file manager is that it classifies instances into their classes.

The third component of a file manager is simple: it allows selection of instances. The only variation between managers seems to be whether they allow multiple selection or single selection only.

The next component is common to all file managers, but has different expressions in each: an action may be performed on an instance of an object. `xdtm` allows the user to select a task situation, and apply any element of the task to the instance. `X.desktop` (like Microsoft File Manager) allows only a single application to apply to an instance. Both Looking Glass and `xmfm` allow a (fixed) range of applications to apply to an instance. Obviously, I prefer the `xmfm` model: it reflects the idea that an object has many methods that can apply to it; the object contains the information about its methods, versus the Microsoft version in that a single application contains all of the methods applicable to a object.

Connection betwen Components

These three components appear to be inseparable: display, classification and selection of instance data. The choice of data in file managers depends on filters: separating this out gives a more general component.

`xmfm` and Looking Glass allow a refinement of instance methods; `xdtm` allows selection of task method; the others only have single action. The general case is: refine a selected instance into a method for the object.

Finally the chosen method on the selected object must be acted on.

A file manager thus consists of four components: data, classification+display+selection, refinement, and action.

The file managers considered so far are monolithic, performing all of these components internally. If they are separated out, how will they communicate? There are heaps of IPC methods. The simple Unix pipeline model decribes the mode apparently needed:

```
ls | classify+display+select | refine | sh
```

Generalisation

Breaking the file manager into components by itself allows a variety of uses. Filtering may be performed by changing the input function:

```
ls -a | ...
ls *.c *.h | ...
```

The person only interested in a graphical display of data can execute

```
ls | classify+display+select
```

The “refine” component can be changed between a single refine as in `filemgr` and `X.desktop`, or a multiple refine as in `xmfm` and Looking Glass. A file manager built out of these components is shown in Figure 2. (In this and later figures, the object selected is highlighted by a surrounding box.

Other Uses

Of more general interest, though, is that the processes of classification and refinement may be configurable. For example, the input data may be a list of names of

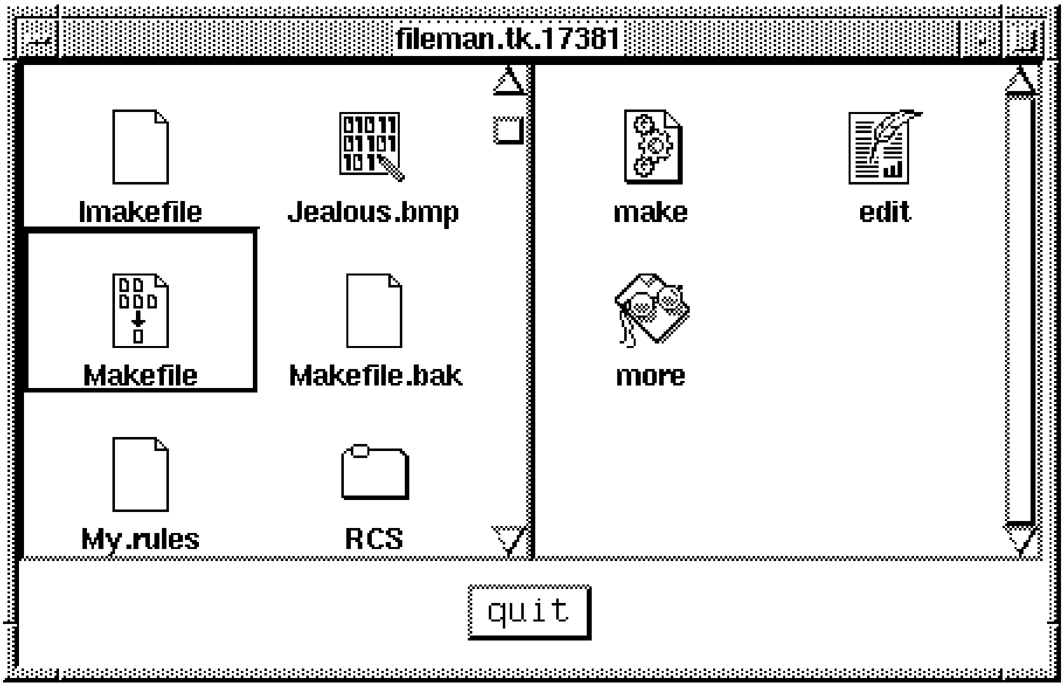

Figure 2: A simple filemanager

people. The classification could be into department, with a symbol typical of that department shown, The classification and display would show names and the department they belong to. Selection of any person within this list could then lead to refinement: choose the work phone number of the person, their home phone number, a map of how to reach them, or other information. The final component (instead of just a shell) would actually perform the appropriate action. This is shown in Figure 3.

A second use is as an aid to language learning. In learning languages with ideographic character sets such as Chinese, one first learns a Romanised system[3]. After this comes the association with the ideographs. Using a classification where a Romanised phrase is matched to its ideograph gives an "ideograph generator," as shown in Figure 4.

Another use is as jukebox. A list of composer names could be the data, with their picture as the icon. Selection of a composer could then bring up a list of their works, and further selection of one of these would send the name of the work to an audio application that would play it. This is shown in Figure 5.

Further Design Issues

Before proceeding to implementation, there are a number of design issues that need to be addressed.

The first of these concerns the eventual granularity of the applications. Each component will need to be a standalone application so that it can be combined with any other applications that the user has. For example, the user may wish to have a filter between selection and refinement, perhaps based on access permissions:

```
ls | classify+display+select | security-filter |
refine | sh
```

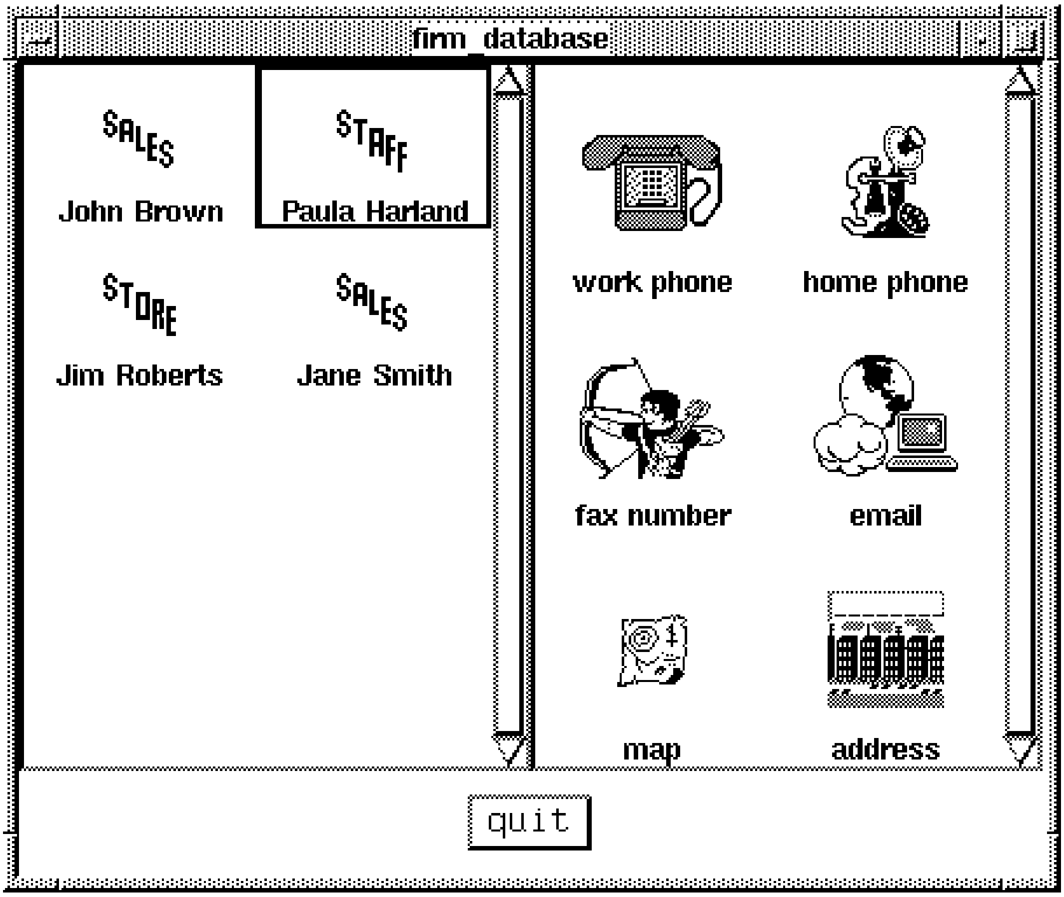

Figure 3: Departmental information

This would be harder to do if each component required a complicated framework before it could be used. The Unix application level seems appropriate.

On the other hand if each graphical component were stand–alone, it would lead to very fragmented interfaces with bits of a composite application all over the screen. Some means needs to be found to link them into higher–level applications.

The second issue concerns the classification and refinement languages. `xmfm` uses shell expresions whereas `X.desktop` and Looking Glass use proprietary languages. For full generality, something with the power of a real programming language is needed, but it should not be a one–off language just for this system.

The third issue concerns the pipeline model. While this works well in many applications, it actually fails for one action common in file managers: changing directories. A change of directory has to be fed to all components, so that the data comes from the new directory, the classification is performed in the right directory (for example, if the classification language uses the type of a file, it must be able to access the file), and the action must be performed in the correct directory. This requires a method to inform all components of the directory change. This is very hard in a pipeline, unless ‘‘special case’’ methods are built into the components to recognise such directory changes. The point is not that it can’t be done, but the methods to solve it for pipelines don’t generalise easily to new situations.

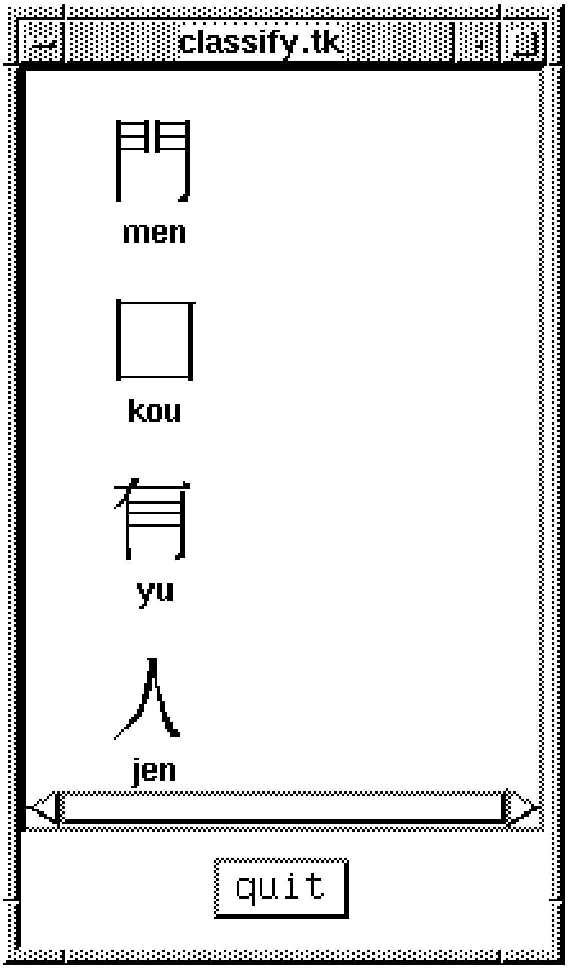

Figure 4: Chinese ideographs from words

Implementation

A prototype of this system was built by breaking the `xmfm` code into the separate bits. This was done using the Xt toolkit with Motif interface. Shell programming expressions were used for the classification language (with the choice of shell made by the user), but for all the other design issues, this choice of system implementation failed.

A full implementation is now under way using Tcl/Tk. The components are all being built as files of procedures which can be incorporated (using the tcl command `source`) into larger applications. To make a standalone component, the file is incorporated into into a simple framework that basically adds a ‘‘quit’’ button. To make composite applications, more than one of these procedure files is sourced.

Either Tcl or the shells allow standalone components to be connected in pipelines. This means that a standalone component should be able to read from standard input. A simple modification to the `wish` interpreter was made so that if the comand name is `iwish` (interactive wish) then it will both read a command line file and then con-

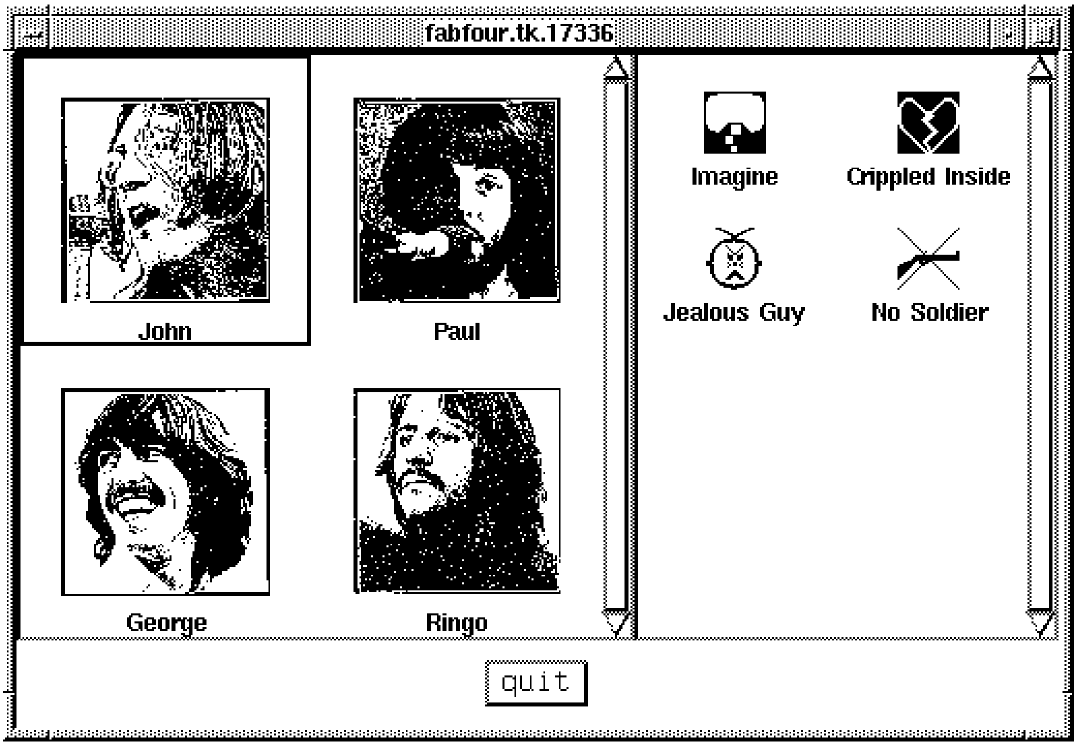

Figure 5: A jukebox

tinue to read from standard input. This is needed for these tools, but is also of general use as it allows command line interaction with any `wish` based system*.

To allow more sophisticated communication than is given by the pipeline, use is made of the Tk `send` mechanism. Each component is built so that its major actions are controlled by tcl procedures. These form the public interface to each component. The lack of a hiding mechanisms in tcl causes problems with this and some ‘‘hiding–convention’’ is needed to deal with this.

By default, the applications read from standard input and write to standard output. The input is expected to be tcl procedure calls that the procedure understands, and the output has the syntax of tcl procedure calls that hopefully can be understood by a later component of the system. In addition, as explained, an application can accept Tk `send` input. To direct the output in a suitable manner, these applications all have a command line option `-sendTo application`. If this is defined, then output is directed to the named application instead of to standard output.

This communication model allows a great deal of flexibility, sufficient to overcome the communication problem for a file manager: the ‘‘action’’ command at the end of the pipeline simply sends a ‘‘cd’’ message and then an application specific ‘‘refresh’’ to each earlier component of the pipeline. Indeed, it even allows one to throw bits of the pipeline away:

*An Internet posting also suggested using `cat file - | wish` This avoids modifications to `wish`, but loses the interactive prompt.

```
ls | classify+display+select -sendTo refine &
refine | sh
```

This could be useful if one wanted many versions of `classify+display+select` to be able to run simultaneously, but at most one copy of `refine`: it could check on the number of other copies running and exit if already running. Each copy of `classify+display+select` would send to the only running copy of `refine`.

The classification and refinement languages are implemented as tcl files that are sourced into the component. Some of the file will consist of commands understood by the component. The `classify+display+select` application defines the procedure

```
classifier ?class ?classifier-function ?bitmap
```

which defines a new class, its recogniser function and its bitmap to the `classify+display+select` application. The configuration file can then contain entries such as

```
classifier Makefile {regexp {^[Mm]akefile$}} \
                         @makefile.xbm
```

which will run the `regexp` command with suitable parameters to recognise the class Makefile with bitmap `makefile.xbm`. Any defined tcl procedure can be used, and any procedures needed that are not already in tcl can be defined in the configuration files. This removes any limits from the configuration languages.

Status of Implementation

Some of these ideas were tested in an Xt/Motif version, which has since been discarded. Modifications have been made to wish so that it behaves as described when it is `iwish`. The two basic applications "`classify.tk`" and "`refine.tk`" have been implemented, although they currently only take options from the command line instead of from the resource database.

This allows simple compound applications to be built such as the address book and jukebox. They can be run with each component as a separate application, or sourced together into a single toplevel window. The file manager requires a more complex "front–end" feeding directory information into the pipeline (or composite application) and a "back–end" that either executes commands or sends chdir information back to the other stages. This has been implemented to the basic functionality of `xmfm`, but without many of the "bells and whistles". These can be added fairly easily.

Problems of Implementation

The lack of modularity in tcl is a problem as it forces the application writer to pay close attention to naming conventions.

The Unix pipeline generally works okay. However, it is not ideal for interactive programs due to the buffering that can exist in I/O buffers and in the pipeline itself. Where applications in the pipeline are line–buffered or raw there is no problem, but some do not allow this buffer control. This then results in a stage "hanging" as it awaits its input. For example, commands such as `sed` can cause buffer delays. In such cases it is sufficient to use a Tk application that flushes each line or uses the send mechanism, but this means that some good Unix tools cannot be used sometimes.

The third problem concerns the `send` primitive. In the file manager application access is needed to this primitive by the front– and back–ends, but they do not require windows. This either leads to redundant `wish` windows or to yet another custom–built Tk interpreter. It would be desirable if some similar communication model was available at the tcl level.

Conclusion

This paper has shown that a typical file manager holds the seeds of a number of other applications and can be easily generalised to encompass them. The choice of implementation framework can have quite an effect on the ease of this, and Tcl/Tk is quite appropriate for building small, reusable communicating applications.

References

[1] J. D. Newmarch, ‘‘XmFm – An X/Motif File Manager,’’ Proc AUUG Conference, 1993 (to appear).
[2] P. Haahr, B. Rakitziz, ‘‘Es: A shell with higher order functions,’’ Usenix Conference Poceedings, Winter 1992.
[3] H. R. Williamson, ‘‘Teach Yourself Chinese,’’ Hodder and Stoughton, 1947