VICTORIA UNIVERSITY OF WELLINGTON

Department of Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 471 5328
Fax: +64 4 495 5232
Internet: Tech.Reports@comp.vuw.ac.nz

A Tcl/Tk Tutorial

Robert Biddle

Technical Report CS-TR-94/4
May 1994

Abstract

Tcl is a small programming language designed to be embedded into other programs to enable extensibility and customisation. Tk is an X window system toolkit providing graphic user interface facilities in connection with Tcl. Both were developed by John Ousterhout at the University of California at Berkeley, and are available free. Tcl and Tk together offer an easy route to graphic user interface programming in the X environment. This document is a practical introduction to using Tk and Tcl, concentrating on how they can be used to quickly develop flexible graphic user interface facilities. Some simple examples are worked through, and suggestions made as to how to proceed further.

Publishing Information

The original version of this tutorial was: Biddle, R., "Graphic User Interfaces Made Easy? A Tcl/Tk Tutorial", In *Proceedings of the 9th Uniforum NZ Conference*, Uniforum NZ, 1992. The original has here been updated to reflect some minor changes in the syntax of Tcl/Tk; some minor errors have been corrected too. A section has also been added at the end of the tutorial outlining some significant recent additions to the language.

Graphic User Interfaces Made Easy?

A Tcl/Tk Tutorial

Robert Biddle
Computer Science
Victoria University of Wellington
`robert.biddle@comp.vuw.ac.nz`

Tcl is a small programming language designed to be embedded into other programs to enable extensibility and customisation. Tk is an X window system toolkit providing graphic user interface facilities in connection with Tcl. Both were developed by John Ousterhout at the University of California at Berkeley, and are available free. Tcl and Tk together offer an easy route to graphic user interface programming in the X environment. This document is a practical introduction to using Tk and Tcl, concentrating on how they can be used to quickly develop flexible graphic user interface facilities. Some simple examples are worked through, and suggestions made as to how to proceed further.

1 Introduction

Tcl (Tool Command language) is small easy-to-use language designed to be embedded into other systems to enable simple yet flexible extensibility and customisation. Tk is an X window system toolkit providing simple graphic user interface and event driven facilities in connection with Tcl. Both Tcl and Tk were developed by John Ousterhout at the University of California at Berkeley, and, like the X window system, are available free for a wide range of Unix based systems.

This tutorial is a practical introduction to Tk and Tcl, and will focus on how they can be used together to very quickly develop flexible graphic user interface facilities. This means the main concern will be Tk, and Tcl will be regarded as a supporting framework. Tk and Tcl are easy to learn, but no practical introduction exists: this document is an attempt to fill that gap. It will be useful to read through this document once, and then to go through the examples again and explore further at an X display. Full details about the language and the toolkit may be found in several documents described at the end of this tutorial.

Tcl is a simple programming language oriented mostly to character string processing. It has one type only, the string, and one data structure only, the array — arrays may be indexed by arbitrary strings. The control structures are conventional, `if`, `while`, and so on, and a simple procedure is also available with value parameters, local variables, and a return value. There is a range of built-in procedures that allow easy manipulation of strings, and lists of strings, including pattern matching and substitution. There is also a range of procedures to facilitate interaction with input/output and the external system in general.

The Tcl language is implemented by an interpreter, so that instructions in the language are acted upon as they are recognised. The interpreter is accessed by a subroutine call, and instructions are passed as parameters and results of the interpretation are then returned. In this way Tcl capability may easily be added to another program, allowing access to the programmable flexibility of Tcl. Moreover, associated facilities are also available by subroutine call that allow *new* procedures to be added to the language recognised by the interpreter. These new Tcl procedures are implemented by procedures specified by the host program. In this way, a host program may be extended by Tcl capabilities, and Tcl capabilities may themselves be extended and customised by the host program.

The Tk toolkit is a set of procedures that extend Tcl in this way with facilities for graphics and event-driven interaction using the X window system.

While Tcl was designed to be embedded in other application programs, it is itself sufficient for many string based programming applications. Moreover, with the Tk procedures, it is itself sufficient for many straightforward graphic user interface applications. For such applications, a very simple program — *Wish* — is available that takes lines of input and passes them to a Tcl interpreter. The Wish program may be used interactively, and so is an excellent way to learn about Tcl and Tk; Wish may also be used to interpret script files of Tcl and Tk statements.

2 Getting Started

In the classic book on the C programming language, Brian Kernighan claims "the first program to write is the same for all languages: print the words `hello, world`". In Tcl, the program is:

```
puts "hello, world"
```

To run this program, you can start Wish, and at its prompt (`wish:`) simply type the line above, and hit return. Alternatively, you could create a file with that line, `hello.tcl`, and then have Wish use that file as a script:

```
% wish -f hello.tcl
```

Either way, the words `hello, world` should appear on the standard output.

In Wish, the graphics extensions of Tk are integrated into the Tcl language: they appear simply as extra procedures that can be evaluated in Tcl — with the graphics and event interaction happening separately. When you start Wish interactively at an X display, the presence of the Tk extensions is shown by a square graphics frame appearing as a separate window. This frame is flexible, and will be controlled as a result of Tk procedures read by Wish, and evaluated by the Tcl interpreter. To explore this, try the following Tcl/Tk program:

```
# hello.tcl -- Hello, World in Tcl with Tk Widgets
label .title -text "Tcl/Tk Greeting Program" -relief raised
label .response
button .greet -text "Greet" -command {.response configure -text "hello, world"}

pack .title
pack .response
pack .greet
```

This program creates three graphics "widgets" within the frame: two "labels" and a "button". Labels simply display some text; buttons also display text, but have a "command" associated with them that is executed if mouse button 1 is pressed within the widget. Pressing the "Greet" button should result in the "hello, world" message appearing in the middle of the frame.

The `label` and `button` procedures create widgets with the name given; configuration settings may follow the name. Complex widgets may be constructed that contain other widgets within them, and for this reason a widget must be specified indicating where in the hierarchy it belongs with a *pathname*. The pathname `.greet` indicates the button is

Figure 1: *The display from* `hello.tcl` *after the "Greet" button has been pressed.*

in the original "root" frame (`.`); if there had been an intermediate frame with pathname `.dialog`, the button within would have pathname `.dialog.greet` showing the relationship. The resulting hierarchy reflects some of the approach of the underlying X structures, but in Tcl/Tk the main issue is simply management of the widget name space.

The `pack` procedure is a "geometry manager": in accordance with certain guidelines, it arranges for widgets to be mapped together. This arrangement is flexible, and will change automatically with the packing and unpacking of widgets, as well as with any significant changes in widget configuration settings. By default, widget specified should be displayed within the parent widget. Widgets are arranged with respect to a flexible "cavity", and by default the specified widget is to be displayed at the top of the cavity.

The state of a widget may be inspected or changed by calling a procedure with the same name as the widget itself. The first argument states the operation required, and the remaining arguments are used for the operation. Each "class" of widget has particular operations, but all have the `configure` operation, which takes the remaining arguments as configuration settings: the button action command above sets the text displayed by `.response`, for example. After typing the above program in interactively, the following could be typed:

```
.response configure -text "how do you do?"
```

This will immediately change the text showing in the middle widget; of course, pressing the "Greet" button will change it back. If the configure operation is not followed by any arguments, a list of all the current configuration settings is returned. Exactly what is included in a widget configuration depends on the class of widget, but most include things like size, colour, border width, relief effects, and so on.

As mentioned earlier, the `button` widget class includes a configuration setting `-command` that specifies what Tcl command to execute if the mouse button 1 is pressed within the widget. Such event handling is available with more flexibility via the `bind` procedure. For example, it is also possible to associate a command with mouse button 2 being pressed:

```
bind .greet <ButtonPress-2> {.response configure -text "bye for now"}
```

Now pressing mouse button 2 within the button widget will result in the text `bye for now` being displayed; and pressing mouse button 1 will result in `hello, world` being displayed again. For any widget, the `bind` procedure is capable of binding any Tcl command with a great variety of events.

The major Tk procedures have now all been introduced: widget create (each procedure being the name of a widget class, like `label`); the display management procedure `pack`; widget specific procedures (each being the name of a widget itself, like `.response`) with specific operations (like `configure`); and the event handling procedure `bind`, which associates any event involving a widget with some Tcl code to be executed.

These few Tk procedures, the availability of a useful set of widget classes, and the flexibility of Tcl programming, are sufficient to easily create a wide variety of programs with useful visual and event driven interfaces. The next steps in learning how to create such programs involve learning more about particular widget classes, a bit more more about the Tcl language, more about interacting with Tcl and Tk programs, and more about about the important display management procedure `pack`.

3 The Tk Widget Set

Because all the Tk facilities are extensions to Tcl, and because Tcl is so easily extended, there is little reason to regard any particular set of widgets as "intrinsic". However, the distribution of Tcl and Tk includes a very useful set of widgets, and it seems reasonable to assume that they, at least, will be available in any Tcl/Tk programming situation. These widgets are described briefly in the following table:

Widget Type	**Brief Description**
`label`	displays a flexibly sized single line of configurable `text`
`message`	displays a block of configurable `text`; flexibly sized to display the entire text in a rectangular block with configurable `aspect` ratio
`button`	displays a flexibly sized single line of configurable `text`; configurable `command` is executed on press of mouse button 1
`checkbutton`	displays a small square-shaped box; operations to change and inspect state and appearance of the box from empty to colour filled; configurable `command` is executed on press of mouse button 1
`radiobutton`	displays a small diamond-shaped box; operations to change and inspect state and appearance of the box from empty to colour filled; configurable `variable` is associated with button state, so states of radiobuttons are arranged mutually exclusive; configurable `command` is executed on press of mouse button 1
`scale`	displays a bar of configurable size with a slider that may be dragged along the bar with the mouse button pressed, with a numeric value displayed alongside, changing with slider motion; the numeric value may be changed with the `set` operation, and inspected with the `get` operation
`entry`	displays a box of configurable `length`; various operations allow text to be inserted, deleted, inspected, scrolled, and selected
`menu`	displays a vertical menu of items displaying lines of text; various operations to insert, delete, and select items, and to associate settings and commands with item selection
`menubutton`	displays a flexible single line of configurable `text`; configurable associated `menu` is displayed ready for item selection on press of mouse button 1
`listbox`	displays a box of configurable size, with a vertical sequence of associated lines of text; operations to insert, delete, and select lines, and to show various sequences of the lines within the box
`scrollbar`	displays a bar of configurable size with arrows and slider; various operations allow position of slider to be changed and inspected; useful in controlling `listbox` widgets

Widget Type	Brief Description
`frame`	holds other widgets within, allowing them to be treated as a unit for display management
`toplevel`	holds other widgets within, and is displayed as a separately managed window; useful for temporary dialogs

4 The Tcl Language

This tutorial is concerned mostly with Tk, and is regarding Tcl as the framework for Tk. The examples presented here show some of the features of the Tcl language, but by no means all of them. There are different variations on the commands shown, and many commands are not introduced at all. But most of the language features are simply useful procedures, and do not require learning special syntax or difficult concepts.

The list and string orientation of the Tcl language is reflected in the language itself: statements and expressions are in fact simply lists of strings. Variable names may be evaluated by prefixing the name with a dollar-sign (e.g. `$inputline`). Strings separated by spaces may be regarded as a whole by surrounding them by double quotes (`"one string"`). Lists of strings may be evaluated, with the first string used as a procedure name and the rest as arguments, by enclosing the list with square brackets (e.g. `[refresh table $inputline]`). This evaluation is done automatically if a list appears as a complete statement, either on a line alone, or separated by a semicolon (`;`). So in fact, all Tcl "commands" are really procedures: even the `proc` command is a procedure call to establish another procedure, with the argument list and body being strings.

While evaluation of variables and procedure calls as arguments must be explicit, the evaluation should not be done too soon. Evaluation of variables and procedure calls may be deferred by enclosing strings within braces; this illustrates the strong string nature of Tcl, as it is the method used to nest control structure:

```
if {$fetch} then { puts [lookup $key] } else { puts $line }
```

To a C programmer, the style looks reasonably familiar, but the braces are there to prevent premature evaluation of the arguments to `if`.

5 Interacting with Tcl and Tk Programs

Some classes of Tk widgets, the buttons, have a configurable `command` setting that associates a Tcl command to be executed when some event concerning the widget occurs, like the mouse button being pressed within the widget.

However, many widgets do not have an association between an event and an action that may be so simply specified. Moreover, even buttons may sometimes need something more sophisticated. Using the `bind` command with buttons to associate different commands with different mouse buttons has been shown earlier, for example.

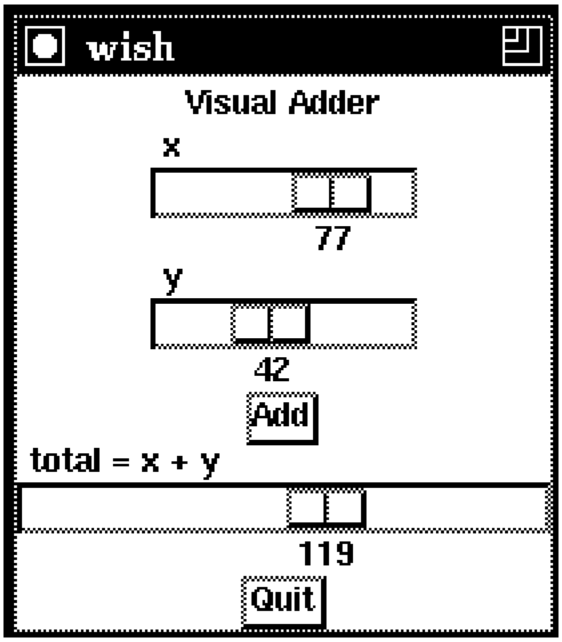

Figure 2: *The display from* `adder.tcl` *showing the value of the bottom scale to be the sum of the values of the two top scales. This can be deferred until the button is pressed, or continous with motion of either of the top two scales.*

Consider the following program, where the settings from scales are only used when a `button` is pressed:

```
# adder.tcl -- Program that Adds Two Numbers Visually
label .title -text "Visual Adder"
scale .x -to 100 -length 100 -label "x" -orient horizontal
scale .y -to 100 -length 100 -label "y" -orient horizontal
scale .total -to 200 -length 200 -label "total = x + y" -orient horizontal
button .add -text "Add" -command settotal
button .quit -text "Quit" -command exit

proc settotal {} {
     .total set [expr "[.x get] + [.y get]"]
}

pack .x; pack .y
pack .add
pack .total
pack .quit
```

In this program, the user may set the values of two numbers using the two top scales. When the "Add" button is pressed, the setting of the bottom scale is set to the sum of the two above.

Now consider the same program, but with the addition of the following two statements:

```
bind .x <Button1-Motion> settotal
bind .y <Button1-Motion> settotal
```

Here any motion of cursor while the mouse button is depressed on the scale slider causes the `settotal` procedure to be called. The display is the same, but now the bottom scale shows the sum of the upper scales — *as they are themselves being set.* This means that with Tk, dynamic input can be reflected dynamically: a powerful method of visual feedback. The flexibility of Tk here is extraordinary — it is even possible to use a dynamic action to cause continous re-configuration of the widget itself involved in the action. Consider this sequence:

```
# self configuring widget...
scale .self
pack .self
bind .self <B1-Motion> {.self configure -width [.self get]}
```

The `bind` command also recognises a wide variety of other events: button press for each mouse button, possibly modified by other keys or repeated, button release, keyboard entry, and cursor motion of various kinds.

The `entry` widget class requires careful attention in keyboard event handling. Not only must characters be inserted as typed, but simple line editing facilities should be catered for. The following program needs to allow a file name to be typed into an `entry` widget:

```
# checkread.tcl -- Check If a File is Readable
label .title -text "Check File Readability"
entry .file -relief sunken
button .checknow -text "Check" -command docheck
label .report
button .quit -text "Quit" -command exit
bind .file <Return> docheck

proc docheck {} {
    if {[file readable [.file get]]} then {
        .report configure -text "File IS Readable"
    } else {
        .report configure -text "File IS NOT Readable"
    }
}
pack .title
pack .file
pack .checknow
pack .report
pack .quit
```

This program allows a filename to be entered, and when the button is pressed it checks to see if the file is readable. In order to be able to type the name of the file into the `entry` widget, the `bind` command be be used to associate various keyboard events with appropriate activity in the widget.

A number of straightforward events are set up by default, and simple typing does cause the appropriate characters to be entered and displayed in the widget as expected. However, an explicit `bind` directive is used here so that the pressing the "return" key causes the procedure `docheck` to be called. In fact, there are many other events one might wish bound to a text entry widget, from re-positioning with the mouse, to a full cut and paste capability.

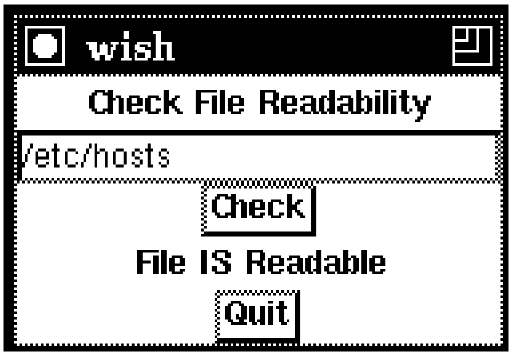

Figure 3: *The display from* `checkread.tcl` *showing the result of pressing the button to check whether a file is readable.*

The best way to deal with complicated lists of bindings you frequently need is to write a procedure that sets up the bindings for any widget you pass as an argument.

The program above also shows Tcl interacting with the Unix system. The `file` procedure provides various information about files, depending on its argument — in this case `readable`. The `open` procedure opens files for input or output, returning a file descriptor string for use with `read` and `puts`.

As well as allowing input and output with the wider system, Tcl allows external programs to be executed:

```
# setbeep.tcl -- Set X Beep Preferences, via xset (1)
label .title -text "Set Beep Preferences"
scale .volume   -label "Volume: %"       -to 100           -orient horizontal
scale .pitch    -label "Pitch: Hz"       -to 20000         -orient horizontal
scale .duration -label "Duration: msec"  -from 10 -to 1000 -orient horizontal
button .set -text "Set" -command setbeep
button .quit -text "Quit" -command exit

proc setbeep {} {
    exec xset b [.volume get] [.pitch get] [.duration get]
}

.volume set 100; .pitch set 5000; .duration set 100

pack .title
pack .volume; pack .pitch; pack .duration
pack .set
pack .quit
```

In this program, three scales may be used to set the preferred sound of the X display's "beep". When the user has set the scales, pressing the "Set" button results in a call to the Tcl `exec` procedure. This takes a sequence of arguments describing an external command, and the arguments to the command. (In fact, redirection and command pipelines are also allowed.) In this particular case, the X command `xset` is used with the scale values to change the sound of the beep.

Sometimes, it doesn't seem sufficient that graphic display programs act only in response to direct user action. Consider the following program:

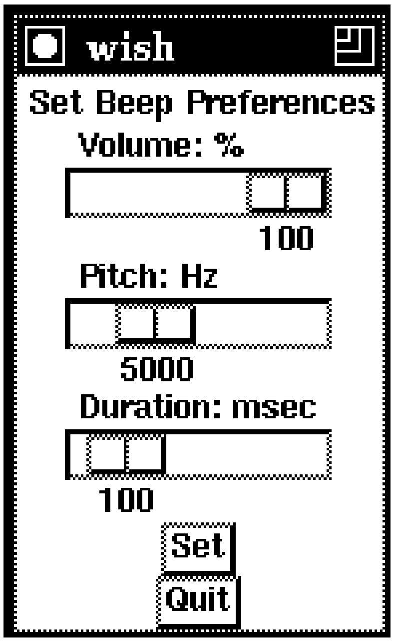

Figure 4: *The display from* `setbeep.tcl` *showing the scales set to change the user preferences to set the X display's beep sound.*

Figure 5: *The display from* `users.tcl` *showing the number of users logged in.*

```
# users.tcl -- Show How Many Users are Logged In
button .checknow -text "Check Number of Users" -command docheck
label .number
button .quit -text "Quit" -command exit

proc docheck {} {
    .number configure -text "[exec who | wc -l]"
}
pack .checknow
pack .number
pack .quit
```

Here, the number of users on the system is displayed whenever the user presses the "Check Number" button. Notice that this uses the `exec` procedure to execute the Unix `who` command, and pipe the output into the Unix `wc` program to count the lines — the resulting output is returned and used to set the number displayed.

If the user wants the number to be updated *automatically*, without having to press the button, the Tk `after` procedure may be used:

```
proc keepchecking {} {
    docheck
    after 60000 keepchecking
}
keepchecking
```

This addition to the above program will cause the number of users to be found out every minute ($60,000$ milliseconds), and the display updated.

6 Display Management with `pack`

Display management using `pack` involves three main tasks: arranging for a widget to be displayed, arranging layout of the displayed widgets, and arranging for widgets to not be displayed.

Specifying a widget *not* be displayed anymore is easy:

```
pack forget .title
```

This results in the widget no longer being displayed. Note that the widget and any configuration settings still remains, and it may be displayed again if so wished. If a widget is no longer necessary at all, the `destroy` procedure may be used:

```
destroy .title
```

After this, the widget no longer exists at all, though a new one with the same name may be created later, and newly configured. Destroying widgets that contain other widgets within them results in the destruction of the inner widgets as well, and all widgets may be destroyed by:

```
destroy .
```

The Wish program interprets Tcl line by line, until end-of-file, but remains in execution until all widgets are destroyed — or the `exit` procedure is called.

Specifying display layout is not so straightforward, because the `pack` procedure involves managing flexible arrangements of a changing set of changing widgets. It is very difficult to perform these tasks without involving very complex specifications, and `pack` attempts remain understandable by limiting the specifications possible.

Essentially, `pack` requires that a display be either a vertical or a horizontal arrangement. This may seem annoyingly restrictive, but there are sources of flexibility within the model. Most significantly, `pack` may be used to arrange widgets within a `frame` widget, and then may be used again to arrange `frame` widgets themselves at a higher level. In this way, the total arrangement might be, for example, a vertical arrangement of horizontal arrangements with vertical components.

```
# setaccess.tcl -- Set File Access via chmod (1)
label .title -text "Set File Access"
entry .file -relief sunken
frame .mode
button .set -text "Set" -command {setmode}
button .quit -text "Quit" -command exit

proc setmode {} {
    global modeflag rflag wflag xflag
    exec chmod $modeflag=$rflag$wflag$xflag  [.file get]
}

frame .mode.class; frame .mode.access

radiobutton .mode.class.user   -text "owner"      -variable modeflag -value "u"
radiobutton .mode.class.group  -text "group"      -variable modeflag -value "g"
radiobutton .mode.class.other  -text "other"      -variable modeflag -value "o"
radiobutton .mode.class.all    -text "every"      -variable modeflag -value "a"
checkbutton .mode.access.read  -text "readable"   -variable rflag -off "" -on "r"
checkbutton .mode.access.write -text "writable"   -variable wflag -off "" -on "w"
checkbutton .mode.access.exec  -text "executable" -variable xflag -off "" -on "x"

pack .mode.class.user  -fill x
pack .mode.class.group -fill x
pack .mode.class.other -fill x
pack .mode.class.all   -fill x

pack .mode.access.read  -fill x
pack .mode.access.write -fill x
pack .mode.access.exec  -fill x

pack .mode.class -side left; pack .mode.access -side left

pack .title; pack .file; pack .mode
pack .set -fill x
pack .quit

.mode.class.user invoke;
.mode.access.read invoke; .mode.access.write invoke
```

The above program arranges a display for a program to change the access mode of a file. There is a vertical sequence consisting of a title, an entry box for the file name to be typed, the mode settings, and a button to set the mode and a "quit" button. The mode settings, however, consist of a horizontal arrangement to specify the class of user accessing, and the mode allowed; vertical arrangements of four and three choices, respectively. This shows, therefore, how more complex graphic layouts may be organised with `pack`. This program contains all the necessary instructions to actually set the file mode with the `exec` call to the Unix command `chmod` in the procedure `setmode`, so this program also shows how useful graphic user interface programs may be developed quite easily using Tk and Tcl. (Though note that this program is really only a simplistic attempt at dealing the many complications

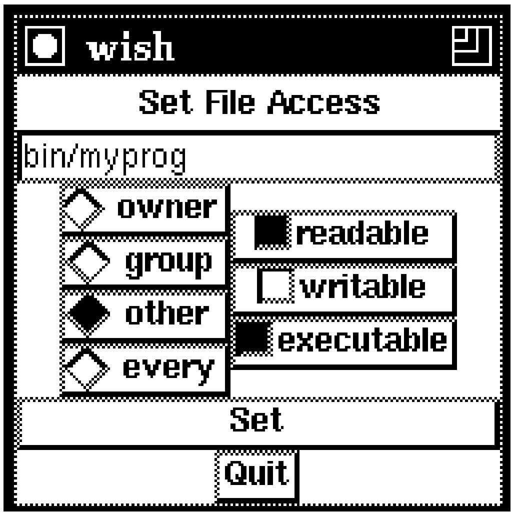

Figure 6: *The display from* `setaccess.tcl` *showing the result of packing small vertical frames within the middle horizontal frame within the vertical outer frame.*

of file access modes.)

As well as allowing the linear arrangements to be nested, `pack` also allows some control over the exact placement of widgets within the linear arrangement. In the example above, each of the widgets is packed at the `top` of the cavity below a vertical arrangement, or at the `left` of the cavity beside a horizontal arrangement. It is also possible to build up the arrangement in similar ways specifying `bottom` and `right`. Moreover, other options may specify that a widget be padded with some space horizontally (`-padx` *n*) or vertically (`-pady` *n*), or that it is to fill the flexible space horizontally (`-fill x`), or vertically (`-fill y`). In the above program, for example, the mode buttons and the set button use `-fill x` to align all the widgets within their vertical frames.

All the `pack` calls discussed so far have specified `append`. The `pack append` construct is fine for building up widget arrangements, putting them in place as they are needed. There are two other options used in connection with displaying widgets, `before` and `after`, that allow a widget to be placed at a specific point in an existing arrangement. While `append` must be followed by a parent and a child widget, both `before` and `after` involve specification of two sibling widgets, one already mapped, and another to be mapped either before or after the first. While `pack append` is usually sufficient for static displays of widgets, `pack after` and `pack before` are useful when displays are being organised in a complex order, or require re-organisation as a necessary part of the display. For an example of how this can be useful, see the VUW CR system performance viewer, described in a later figure.

7 Program Development with Tcl/Tk

All of the programs and program fragments presented here have been purely Tcl and Tk — with occasional `exec` calls to Unix. Quite useful programs can be constructed in this way. In particular, Tcl/Tk programs can be ideal to construct helpful graphic user interfaces to existing Unix commands and programs, as shown above. Such interface programs can use the various widgets of Tk to make it easy for users to specify various settings and options which otherwise require learning of difficult command syntax. Moreover, the list and string processing capabilities of Tcl, together with the communication procedures, allow

Unix commands to be run from Tcl, and for their output to be retrieved and analysed. All this is especially useful for commands and programs that are not used frequently, and that have complicated syntax: the file protection mode changer `chmod`, and the X preferences command `xset` have been briefly discussed, and the file archiver `tar` and file locater program `find` are also in this category — and there are many others.

Programs that just use Tcl and Tk can simply use the Wish program itself as a command interpreter, and like Shell scripts can be made executable with a first line specifying this:

```
#!/usr/local/bin/wish -f
#This file may now be made executable and will be interpreted by Wish
#(make sure the wish pathname above is correct for your system)
.label .title -text "My Wonderful Tcl/Tk Program"
...
```

When such programs are still in development, it is often useful to use Wish interactively, and use command `source myprog.tcl` to include the program source. In this way the program will be run, but you may still give commands interactively to Wish to explore the program, printing values, reconfiguring widgets, as necessary. The `source` directive is also useful for including sets of useful procedures so they are available for new programs.

Both as an aid to exploring program behaviour, especially event driven behaviour, as well as debugging, the Tcl command `trace variable` is also very useful. It can be used to associate a given variable with a command to be executed every time the variable is accessed. For the `chmod` program shown earlier, the following setup would print helpful lines every time the variable `modeflag` is set (`w`) or inspected (`r`):

```
proc mydebug {name arrayindex usage} {
    puts stdout "Variable $name: Usage: $usage"
    puts stdout "Value is now [expr $$name]"
}
trace variable modeflag rw mydebug
```

Where programs use `exec` to invoke external Unix programs, the Tcl command `catch` is useful for *handling* errors or unexpected results. Normally any failure of an external command results in an error message and call backtrace being printed on standard output. The `catch` command will successfully invoke *any* command, and return the completion code.

Some advanced features of Tk have not been introduced here, but are worth at least mentioning. For instance, it is useful to know that widgets created using Tk can work together with X "resource" facilities to integrate configuration of Tk programs with other X clients. There is also a Tk command `wm` that allows direct communication with the window manager. And lastly, Tk presents an extremely easy-to-use method for arranging communication between Tk applications: the simple `send` procedure arranges for a Tcl command to be executed by the interpreter of a *different* Tcl/Tk application, even if it is running on a different machine. This suggests that a well designed set of applications could offer interesting capabilities by working together: an opportunity that seems worth pursuing.

And while the approach discussed above does allow graphic users interfaces to be constructed very easily, it is important to remember that Tcl and Tk do offer much more flexibility if necessary. New programs can be written that use the ability to work *with* Tcl so that the program can use the Tk facilities for all user interaction from the original design. Whereas the approach described earlier involves using Tcl to *externally* drive existing programs, this approach would involve working with Tcl *internally*, with the host program driving the Tcl component. Further consideration of this approach is beyond this discussion, but the documentation described later provides all the details necessary. A good way to start is to closely examine the C source to the Wish program itself.

8 The VUW Meters Widget Set

At the Computer Science department at Victoria University of Wellington, our work in management of distributed systems requires various graphical displays to allow easy comprehension and comparison of empirical data. Especially because we wish to explore various approaches in data display, the flexibility of Tcl and Tk appeared very useful. Accordingly, we have developed widget classes within the Tcl and Tk model specifically for display of empirical data; we call them "meters". Some examples of these are described in the table below:

Widget Type	Brief Description
`bargraph`	shows a varying length rectangular bar against a set scale; length, width, range, scale, orientation, and colour are configurable; operations `set` and `get` the current value
`dial`	shows a varying needle set against a scaled circular dial; radius, range, scale, dial portion, and colour are configurable; operations `set` and `get` current value; optional radii mark extrema from recent settings
`stripchart`	shows a moving sequence of varying height bars against a set scale, indicating the recent values set; length, width, range, scale, colour, and number in the sequence are configurable; operations `set` and `get` current value

All the widgets in this set may be used in a similar way to other Tk widgets, and so created, configured, and arranged from Tcl. They may also be set and inspected from Tcl, and so can be used when better empirical display would improve the graphic interface. Consider the following modification to the program presented earlier to show the number of system users:

```
# usersdial.tcl -- Show with a Dial How Many Users Logged In
button .checknow -text "Check Number of Users" -command docheck
dial .number -maxvalue 30 -numticks 30
button .quit -text "Quit" -command exit

proc docheck {} {
    .number set [exec who | wc -l]
}
pack .checknow
pack .number
pack .quit

proc keepchecking {} {
    refresh
    after 60000 keepchecking
}
keepchecking
```

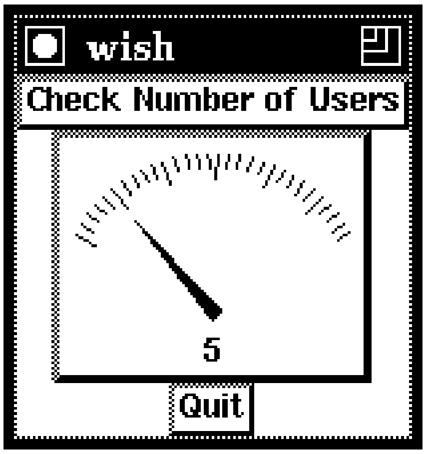

Figure 7: *The display from* `usersdial.tcl` *showing the number of users logged in. Note that this is really more cluttered than necessary, because with* `bind` *the dial itself can play the role of a button.*

Even simple programs like this are quite useful, and can be used to provide clear visual indicators of important varying empirical quantities: free space on critical disk file systems, for example, by using the Unix `df` command to provide the data.

Tcl is interpreted statement by statement, however, and that is a liability where efficient real-time monitoring of live data is required. In addition to the conventional Tcl/Tk use of these widgets, therefore, a lower level of interaction is also provided. As described in the introduction, Tcl is designed to be imbedded into programs to provide flexible and programmable facilities, and Wish is just a simple program that passes input lines to the Tcl interpreter. Like all the Tk widgets and related procedures, the VUW meters are specified by C data structures and functions that are associated with Tcl using the methods Tcl provides to extend itself to suit the host program.

The VUW meter widgets, however, themselves allow *other* functions to be specified to provide values to set the widget. The widgets include operations to `start` and `stop` this background activity, and a configurable `interval` setting that determines how often it takes place. In this way, we have access to the great flexibility of Tcl and Tk for widget arrangement and user interaction – where the interpreted nature of Tcl is an advantage and presents no impact on overall performance. But we also have access to the low levels to efficiently update values very frequently so we can display in real-time live data from various sources in the system.

9 Conclusion

Tcl and Tk are simple in design and structure, though of course they do require some exploration to become familiar with the way they work. However, the interpreted nature of Tcl allows instant feedback, and makes such exploration easy and fun. From a software engineering point of view, the whole approach has a very high power–to–weight ratio.

The nature of the Tcl approach offers great variety in application. In developing interfaces to existing programs and commands, the string orientation and easy access to Unix files and commands are valuable. In developing more sophisticated systems, the ability to embed Tcl in another program is very useful. And the design of Tcl and Tk as an *interpreted* language allows an extraordinary flexibility in what can be achieved, as well as providing the

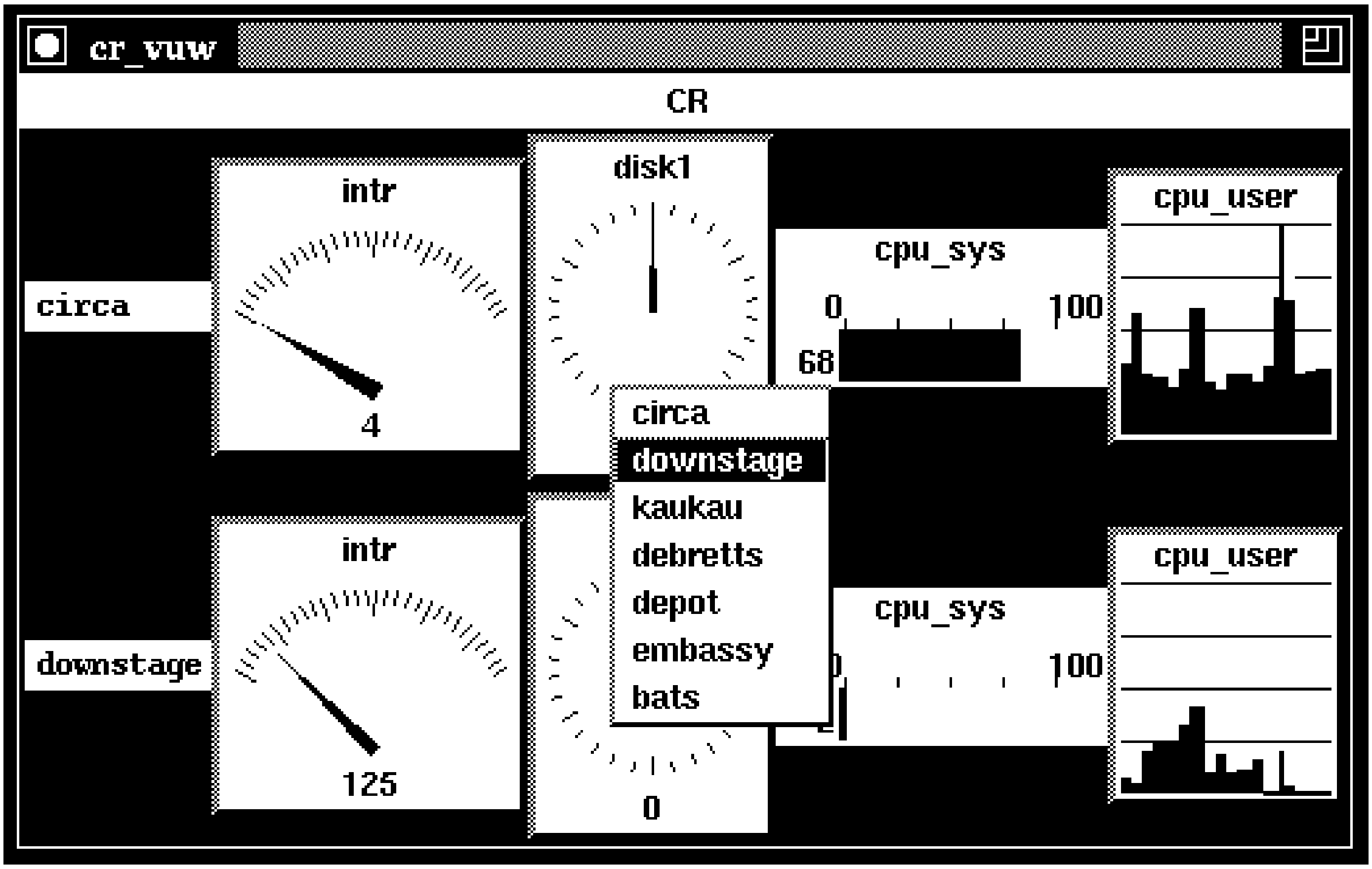

Figure 8: *The display from the VUW CR (Control Room) distributed system performance viewer. Each row represents data from a particular machine, and each column a particular performance metric, allowing easy visual comparison. Any number of rows and columns may be displayed, and the machines and metrics are chosen by pop-up menu, and placed at the cursor position. The configuration of the meter for any metric may be changed while viewing. Live data is fed to the meters at a low level from other monitor processes.*

opportunity to explore while developing.

The approach is still new, and much more experience is needed in using these tools in a wide range of situations. There may well be subtle problems with some of the simplifying assumptions made — though the design is still evolving. However, it does appear that Tcl and Tk at least open the door to easy programming of graphic user interfaces in the Unix and X window system environment.

10 Where to Learn More about Tcl and Tk

The original documents describing Tcl and Tk are by their creator, John Ousterhout:

> Ousterhout, J. K., *Tcl: An Embeddable Command Language*, in Proceedings of the Winter Usenix Conference, 1990.
>
> Ousterhout, J. K., *An X11 Toolkit Based on the Tcl Language*, in Proceedings of the Winter Usenix Conference, 1991.

These two papers provide a formal introduction to the aims and design philosophy behind Tcl and Tk, make comparisons with related work, and indicate future directions for development. They are not, however, suitable as detailed reference material for programming with Tcl and Tk.

For details about programming, the main reference source is the set of Unix manual section 3 entries supplied with the Tcl/Tk distribution. These are well written at the detail level, and reasonably complete. In some distributions, the definitive source of information on the Tcl language itself is the entry `tcl (3)`. This is more than 30 pages long typeset, and explains the syntax of the language, as well as all the builtin procedures useful in string and list processing, and interaction with the system. In other distributions, the various features of Tcl are in separate manual entries.

In some distributions, various kinds of command manual entries are distinguished by different prefixes. For the Tk commands available directly in Tcl, the documentation is in a set of entries such as `tk_label`, `tk_button`, `tk_pack`, and so on — `tk_` followed by a name in lower case. For the Tcl facilities available to a host program written in C, see the entries like `tcl_CrtCommand`, `tcl_Eval`, and so on — `tcl_` followed by a capitalised name. For the Tk facilities available for a host program written in C, see the entries like `tk_ConfigWind`, `tk_GeomReq`, and so on — `tk_` followed by a capitalised name. Other distributions use the same entry names, but without the prefixes.

The Tcl/Tk distribution includes all documentation, as well as all source in C, and a small set of demonstration programs, and is available free. In New Zealand, the distribution is available from the Computer Science department of Victoria University of Wellington: if on the Internet, use *ftp* to connect to `ftp.comp.vuw.ac.nz` and see the directory `/pub/languages/tcl`; otherwise contact the department for advice.

A Usenet news group `comp.lang.tcl` has existed since the beginning of 1992, and is a good source of news and advice about Tcl and Tk.

11 Postscript (May, 1994)

The original version of this document was written in April 1992, and it introduced Tcl/Tk as it was then. Since that time the language has evolved, and the orginal version has here been updated to reflect some minor changes in syntax (principally concerning `pack`); some minor errors have also been corrected.

More significant changes to Tcl/Tk programming have taken place as a result of additions to the language. Because Tcl/Tk is so easy to extend, this is something that will always be happening. However, a few additions to the standard set of facilities are especially noteworthy. None of these need be detailed in an introductory tutorial such as this, but they all need to be briefly explained:

- The `place` command provides an alternative approach to the geometry management of `pack`. Instead of building up and maintaining a rectilinear tiling of widgets, `place` allows widgets to be positioned according to absolute or relative coordinates in a parent widget. This allows the interface designer more explicit control over the particular layout of a window.

- The `text` widget is a major new kind of wigdet for presentation and selection of text. It is capable of handling large amounts of text, and so is suitable for displaying whole documents. Not only can regions of the text be easily selected and modified, but regions of the text can also be bound to events: this enables some "hypertext" behaviour.

- The `canvas` widget is a major new kind of widget for displaying and allowing interaction with pictorial structures. Pictorial items available include rectangles, ovals, lines, polygons, arcs, and bitmaps. These items can be moved, scaled, reconfigured, and bound to events. Canvas widgets are therefore ideal for creating interfaces designed upon a diagramatical metaphor, and are also suitable for creating new composite widgets.

The essential design of the two new major widgets is discussed by Ousterhout in the following paper:

> Ousterhout, J. K., "Hypertext and Hypergraphics in Tk", In *Proceedings of the 7th Annual X Technical Conference*, MIT X Consortium, 1993.

Also, a tutorial for using the canvas widget is now available:

> Biddle, R. L., "GUI Display of Data with TCL/Tk: Using the Tk Canvas Facility", In *Proceedings of the 11th Uniforum NZ Conference*, Uniforum NZ, 1994.

This is also available from the Department of Computer Science, Victoria University of Wellington, as Technical Report 94/5.

The most important addition to TCL/Tk to take place in 1994 will be the publication of Ousterhout's comprehensive book describing the language:

> Ousterhout, J. K., *Tcl and the Tk Toolkit*, Addison-Wesley, 1994.