Prototyping in Tcl/Tk

by SAM
sam@cs.vu.nl

version 1.01

August 13, 1996

1 Introduction

Tcl is an interpreted scripting language, while Tk is the graphical toolkit that extends Tcl with the possibility to create graphical user interfaces. The combination forms a powerful tool for prototyping. Because Tcl/Tk is interpreted, it's possible to edit a program while it's being executed, thereby providing a means to quickly develop a prototype.

This document will give an introduction to Tcl and Tk and will show how Tcl/Tk can be used for creating a simple prototype.

2 An introduction to Tcl

This section will only describe a very small portion of the complete Tcl scripting language, because not much knowledge about Tcl is needed for building prototypes. For a more complete description see [1].

2.1 The interpreter

The Tcl-only interpreter is called `tclsh`, but we will only use Tcl in combination with Tk, and therefore we don't use `tclsh`, but **wish**.

wish is simply started with

```
$ wish
%
```

When **wish** is started, the **wish** prompt (`%`) and an empty window appear. Commands are typed at the prompt and results of Tcl/Tk commands will mostly become visible in the window.

The interpreter is exited by typing `exit` or by pressing Control-C.

2.2 Commands

An example of a simple command in Tcl:

```
% set myvar 27
27
```

A command always starts with the name of the command (in this case `set`), followed by zero or more arguments. Everything is seperated by spaces. The 'result' of a command is always printed.

2.3 Variables

The command in the previous example set the variable `myvar` to **the string** "27". The only data type in Tcl is a string. However, some commands will interpret arguments as numbers, for example the `expr` command (`expr` will be explained later).

To use the value of a variable, you prepend it with `$`:

```
% puts $myvar
27
```

(`puts` is the command to print some value.)

String values are normally delimited with double quotes, but a string of a single word doesn't need them:

```
% set myvar "hello world!"
hello world!
% puts $myvar
hello world!
% set myvar hello
hello
% puts $myvar
hello
```

The value of a variable can be used when defining a string by prepending it with a `$`:

```
% set myvar "world!"
world!
% set myvar2 "hello $myvar"
hello world!
% puts myvar2
hello world!
```

To include a literal '$' in a string, use `\$`:

```
% set myvar "hello\$world!"
hello$world!
```

Variables are not declared, but come into existence as soon as they get a value:

```
% puts $unknownvar
can't read "unknownvar": no such variable
% set unknownvar "known"
known
% puts $unknownvar
known
```

A variable can be destroyed by *unsetting* it:

```
% puts $unknownvar
known
% unset unknownvar
% puts $unknownvar
can't read "unknownvar": no such variable
```

2.4 Expressions

An example of using `expr`:

```
% expr 3 * 27
81
```

`expr` computes the result of the computation formed by its arguments. `expr` doesn't care if you seperate all its 'arguments' with spaces:

```
% expr 3*27
81
```

Tcl supports all C-operators and a number of math functions like `abs`, `cos`, `floor`, etc.

2.5 Substitutions

We already saw one form of substitution: variable substitution by putting `$` in front of a variable. Another form of substitution is *command substitution.* Suppose you want to give to some variable a value that's the result of some computation: that's not possible with the constructs described so far:

```
% set myvar 3 * 27
wrong # args: should be "set varName ?newValue?"
% set myvar 3*27
3*27
% set myvar expr 3*27
wrong # args: should be "set varName ?newValue?"
```

The first and third try will fail because the number of arguments to `set` is wrong. The second try is not so strange when you remember that Tcl only knows strings. What you want to do here is to substitute the value of a computation (for example `expr 3*27`) for one of the arguments. This is done by using square brackets ('`[`' and '`]`'):

```
% set myvar [ expr 3*27 ]
81
% puts $myvar
81
```

Tcl offers a couple of other forms of substitution, but these are not described here.

2.6 Procedures

An example of a simple procedure:

```
% proc sayhi {} {
        puts "hello world!"
        puts "testing 1 2 3"
}
```

Yes, that's it. `proc` means that we are going to define a new procedure. Next is the name of procedure (`sayhi` in this case), after that the names of the parameters (`{}` means there are no parameters) and at last the procedure body. `sayhi` can now be used like any other command:

```
% sayhi
hello world!
testing 1 2 3
```

A procedure with some parameters:

```
% proc myproc {var1 var2} {
        puts "var1 is $var1"
        puts "var2 is $var2"
}
% myproc hello world
var1 is hello
var2 is world
```

A procedure returning a value:

```
% proc multiply {var1 var2} {
        return [ expr $var1 * $var2 ]
}
% puts [ multiply 3 27 ]
81
```

All parameters are passed by value and can't be changed.

2.7 Control flow

The most commonly used control flow constructs of Tcl:

break
: Abort innermost containing loop command.

continue
: Skip to the next iteration of innermost containing loop command.

exit [*returnCode*]
: Terminate the process, returning *returnCode* (an integer which defaults to 0) to the system as the exit status.

for *start test next body*

Looping command where *start*, *next*, and *body* are Tcl command strings and test is an expression string to be passed to **expr** command.

Example:

```
for {set i 0} {$i < 10} {incr i} {puts $i}
```

(This example makes use of *deferred substitution*; everything between {} is passed unsubstituted to the **for** command.)

if *expr1* **then** *body1* [**elsif** *expr2* **then** *body2* ...] [**else** *body2*]

If expression string *exptr1* evaluates to true, Tcl command string *body1* is evaluated. Otherwise if *expr2* is true, *body2* is evaluated, and so on. If none of the expressions evaluate to true then *bodyN* is executed.

Example:

```
if [expr $i == 27] then {puts "i = 27"} else {puts "i != 27"}
```

return [*string*]

Return immediately from current procedure with *string* as return value.

while *test body*

Evaluates the Tcl command string *body* as long as the expression string *test* evaluates to true.

Example:

```
set i 0
while {$i < 10} {incr i; puts $i}
```

(This is another example of using deferred substitution.)

2.8 Executing Tcl-source

One feature of Tcl is very handy when building prototypes: executing Tcl source that resides in some file. For example:

```
% source tcltest.tcl
```

will execute the Tcl commands contained in the file `tcltest.tcl`. Putting separate parts of a prototype in seperate sourcefiles will make your project very flexible.

2.9 Other features

Tcl contains a whole bunch of other features, many of them are very useful, but you probably won't need them when you're building a simple prototype, so these features are not described here. [1] contains a complete list of all features of Tcl.

3 An introduction to Tk

Tk contains a lot of widgets, most of them are not described here. But with the information provided in this document, it shouldn't be hard to be able to use these widgets.

3.1 Tk is event based

Tk is event based. This means that everything that happens is a reaction to an action of the user. Most of the time, the program is in some suspended state, waiting for the user to give some input. When the user gives some input, it is called an event. It's not known in advance what kind of event that will be; the user can pull down a menu, push a button, push another button, scroll a part of a window, enter some value in an entrybox or something else. Each important event should be handled by some piece of code. Telling Tk which code to handle which event is called *binding* of code to events. Event based programs require an other way of thinking and programming than a serial program does.

3.2 Widgets

The term *widget* has been used in the introduction of this section. What exactly is a widget? It's hard to give an exact definition, but 'something that can be put on a window' is very close. Some examples of widgets are buttons, scrollbars, labels, listboxes, menu's, checkboxes ... A widgets may or may not accept input from a user; a button certainly does accept input, a label does not, it only shows some information. But all widgets are a part of the entire user interface.

3.3 Creating widgets

A button can be created in the following way:

```
% button .b -text "My first button"
.b
```

`button` is the command for creating a button widget, `.b` is the *path* or the name of the button (more about paths later). The arguments `-text "My first button"` tell Tk to put the string "My first button" in the button. Now we tell Tk to put this button in the main window:

```
% pack .b
```

And voilà, a button appears in the main window with the text "My first button". Pressing the button won't do anything, because we haven't told Tk what to do when the button is pressed. We can do that in the following way:

```
% .b configure -command { puts "My first button has been pressed" }
```

Pressing the button will now result in printing the string "My first button has been pressed". We have *bound* the command `puts "My first button has been pressed"` to the "push button" event.

3.4 Changing a widget

What exactly did we tell Tk with the previous command? When the button was created, a new command was made with the same name as the button. This command can be used to change things of the button after it has been created. `.b configure` tells Tk we want to change some option of the button named `.b`. The `-command` option is the option that tells Tk what to do when that particular button is pressed. Other options can change the text in the button, change the background color, change the dimensions and a lot more.

3.5 Destroying widgets

A widget is destroying by using the destroy command of Tk:

```
% destroy .b
```

When a widget is destroyed, it is directly removed from the window.

3.6 Packing widgets

When a widget is created, it is not yet visible on the window. Only by *packing* on a window or some other widget will a widget become visible. We've seen simple examples of packing before in sections 3.3 and 3.9. The packer is used to tell Tk where to place a widget, relative to other widgets. Suppose we want one button on the right side of a window and two buttons at the right side, as in figure 1:

```
% button .b1 -text "Button 1"
.b1
% button .b2 -text "Button 2"
.b2
% button .b3 -text "Button 3"
.b3
% pack .b1 -side left -fill both
% pack .b2 -side top
% pack .b3 -side bottom
```

Figure 1: Example of using the Tk packer, using filling

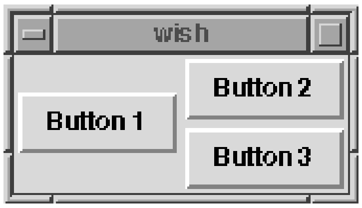

Figure 2: Example of using the Tk packer, without filling

The `-side` argument is used to indicate on which side of the remaining space the widget is to be placed, the new remaining space will be the other side. The `-fill` argument is used to let a widget take all available space (compare figure 1 to figure 2).

The exact workings of the Tk packer are a bit obscure and are explained in detail in [2].

3.7 Binding actions to events

Globally, there are two ways to bind an action to an event. The first way is with the `-command` option of a widget, for example, if `.b` is a button

```
.b configure -command { puts "Button .b is pressed" }
```

will result in the text "Button .b is pressed" printed out every time the button is pressed. The `-command` option is used to bind an action that is to be invoked when a widget is invoked. Not all widgets support this kind of binding, for example a listbox can not be 'invoked'.

The second way will work with almost all widgets and handles events that are generated by the X Windows system (or whatever graphical environment you're running). To bind this kind of events, the `bind` command is used, which has the following syntax:

bind *widget-name* *<event>* *script*

Suppose we want to bind the event of releasing the middle mouse button when the mouse cursor is positioned over button `.b` to an action that consists of printing some text. That could be done as follows:

```
bind .b <Button2-ButtonRelease> { puts "Button 2 released over .b" }
```

Almost every event that the X Window system generates can be bound to an action. Among the most useful events are pressing and releasing a mouse button, moving the mouse cursor, pressing and releasing a key and entering and leaving a widget with the mouse cursor. See [1] for the complete syntax of the events and the `bind` command.

3.8 Frames

A *frame* can be thought of as a window within a window. It acts as a container of other widgets. Frames are needed because the mechanism of the Tk packer alone is too limited to even create user interfaces of little complexity. For example, the application in figure 3 can not be created with the packer only (left as an exercise for the reader).

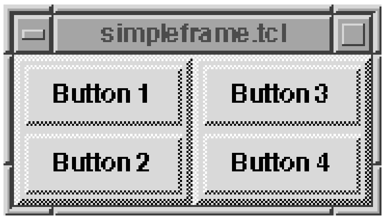

Figure 3: Example of using frames

The commands used for creating this application are:

```
frame .f1 -relief raised -borderwidth 3
button .f1.b1 -text "Button 1"
pack .f1.b1 -side top
button .f1.b2 -text "Button 2"
pack .f1.b2 -side bottom
pack .f1 -side left

frame .f2 -relief raised -borderwidth 3
```

```
button .f2.b3 -text "Button 3"
pack .f2.b3 -side top
button .f2.b4 -text "Button 4"
pack .f2.b4 -side bottom
pack .f2 -side right
```

A frame is created just like any other widget. The widgets that are to be contained in a frame have the path of the frame prepended to their own path. The widgets within a frame are packed into the frame, the frame itself will be packed in a window.

3.9 Widget paths

All widgets in an application form a hierarchy. The path of a widget is the path you have to follow from the root of the hierarchy to that widget.

Consider the hierarchy of figure 4, which is the hierarchy belonging to figure 3. (The reason why the text of button 2 is not visible is that there's just no room since button 3 and button 4 are taking all the available space.)

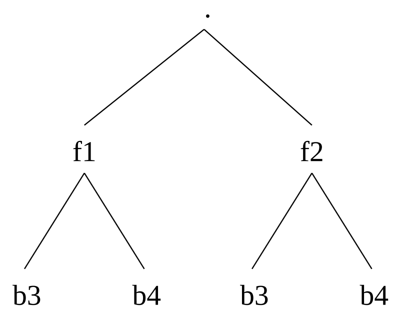

Figure 4: Hierarchy of a simple Tcl/Tk application

The path of a widget determines the widget 'tree' it is contained in. In the example, the paths of button 3 and button 4 are `.f2.b3` and `.f2.b4` respectively, thus both being contained in frame 2.

3.10 Creating windows

So far we've only seen examples of using widgets in the main window of **wish**. But what if we want to create an application that has multiple windows? Then we need to create our own windows, called *toplevels* in Tk. A toplevel can be created like a widget:

```
% toplevel .t -width 320 -height 200 -text "A new window"
```

In contrast to a widget, a toplevel is shown directly after it is created, since there's no need to put it somewhere in a window.

An example of a window using a frame and a couple of buttons is created with the following commands:

```
toplevel .t
frame .t.f
button .t.f.b1 -text "Button 1" -command { puts "Button 1 pressed" }
button .t.f.b2 -text "Button 2" -command { puts "Button 2 pressed" }
pack .t.f.b1 -side top
pack .t.f.b2 -side bottom
pack .t.f -side left
button .t.b3 -text "Button 3" -command { puts "Button 3 pressed" }
pack .t.b3 -side right -fill both
wm title .t "A new window"
```

This creates a window with pathname `.t`. All widgets that are to be put in this window are prepended with this pathname. This example also demonstrates binding simple commands to buttons and changing the title of a window by means of the window manager command (`wm`). To change the title of a window, a command of the form `wm title` *windowpath newtitle* is given (see [1] for more information about `wm`-commands).

The resulting window is shown in figure 5.

Figure 5: Example of a toplevel

4 Building the interface

The section will explain how a prototype is designed by building an example user interface from scratch. We will build the user interface of a file-selector. The most easy way to go about building an interface is to create a script which can be run with `wish -f` *script*, or executed from within **wish** with `source` *script*.

4.1 Outline

First we decide on the global outline of the user interface. Each window is sketched and widgets are globally placed. See figure 6.

4.2 Partitioning in frames

We must be able to pack all widgets on the screen and since this is most of the time not possible without using frames, a partitioning in frames is made. Keep the mechanism of the packer in mind when dividing a window in frames [2]. One possible way of dividing the window for the file-selector is shown in figure 7.

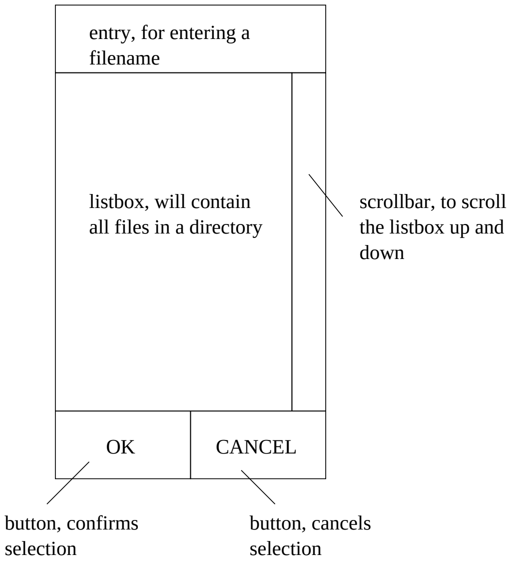

Figure 6: Global design of a file-selector

4.3 The basic interface

When this is done, the first Tcl-script can be built:

```
# create new window
toplevel .fileselect

# entry-frame
entry .fileselect.entry
pack .fileselect.entry -side top -fill x

# filelist-frame
frame .fileselect.files
listbox .fileselect.files.list
pack .fileselect.files.list -side left -fill both -expand true
scrollbar .fileselect.files.scrolly
pack .fileselect.files.scrolly -side right -fill y
pack .fileselect.files -side top -fill both -expand true

# button-frame
frame .fileselect.buttons
button .fileselect.buttons.ok -text OK
pack .fileselect.buttons.ok -side left -fill x -expand true
button .fileselect.buttons.cancel -text CANCEL
pack .fileselect.buttons.cancel -side right -fill x -expand true
pack .fileselect.buttons -side top -fill x
```

The `-fill` and `-expand` parameters are used to let the widgets fill the available space and to let them grow when the window is enlarged (compare this to what happens when these parameters are left out).

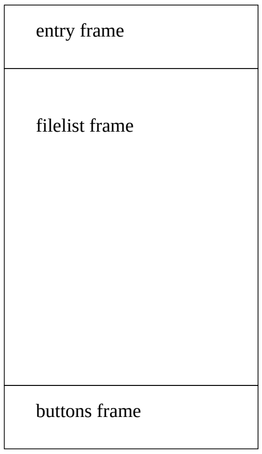

Figure 7: Previous design partitioned in frames

4.4 Adding some functionality

The building blocks of the interface are now in place, but it would be nice if some basic behavior can be specified; for example, we would like to see the scrollbar in action or be able to make a real selection from the list.

With the final file-selector (of which this is the prototype), files can be selected in two ways. The first way is to select a file from the list and click **OK**, the second is to enter a filename in the entry-box and click 'OK'. If **CANCEL** is chosen, nothing should be selected. When either button is pressed, the file-selector should vanish. In the prototype, when **OK** is pressed, the selected file will be printed on stdout, so we can check if the file-selector is functioning properly.

Since this is a prototype, it doesn't really matter if we create a real list of files or make one up. The first is certainly not impossible in Tcl, but for simplicity's sake, we'll take the latter approach. First, we associate the scrollbar with the listbox and vise versa:

```
.fileselect.files.scrolly configure -command {.fileselect.files.list yview}
.fileselect.files.list configure -yscrollcommand \
    {.fileselect.files.scrolly set}
```

The first command tells Tcl that if the scrollbar is changed (for example by clicking on one of the arrows), that the changes are passed to the listbox, which will then update its contents. The second command states that if the contents of the listbox are scrolled up or down, the change should be reflected by the position of the slider.

Next, some data is added to the listbox:

```
.fileselect.files.list 999 file0
.fileselect.files.list 999 file1
  :
.fileselect.files.list 999 file18
.fileselect.files.list 999 file19
```

The listbox now shows a list of 20 filenames and the contents of this listbox can be scrolled by using the scrollbar.

We now define the procedures that are called when a button is pressed:

```
proc do_ok {} {
    if [expr [ llength [ .fileselect.files.list curselection ] ] > 0 ] then \
    { \
        puts [ .fileselect.files.list get \
                [ .fileselect.files.list curselection ] ] \
    } \
    else \
    { \
        puts [ .fileselect.entry get ] \
    }
    destroy .fileselect
}

proc do_cancel {} {
    puts "File selection canceled"
    destroy .fileselect
}
```

The first procedure should be called when the **OK** button is pressed. It checks if there is some selection made in the listbox and if there is, prints the selected item. It there is no selection made, the contents of the entry are printed. Either way, the file-selector window is removed. The second procedure should be called when the **CANCEL** button is pressed. It just prints a message that the selection is canceled and destroys the window.

These procedures can now be bound to the buttons:

```
.fileselect.buttons.ok configure -command do_ok
.fileselect.buttons.cancel configure -command do_cancel
```

This will complete the file-selector. Of course, the final version must have much more functionality, but this is only a prototype; it shows how the file-selector will 'look and feel'. See figure 8 for the 'look'. The final script can be found in `/home/se/doc/tcltk/source/fileselector.tcl`.

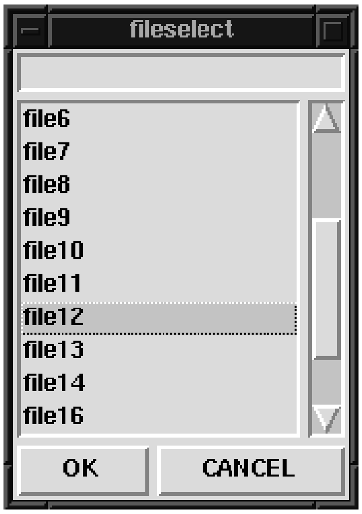

Figure 8: The file-selector prototype in action

4.5 Building the interface, a summary

- Determine what the interface should look like and what widgets are placed where.
- Partition each window in manageable frames. Each widget in a frame will be packed inside its frame and each frame will be packed inside its window.
- Write a script that puts all widgets on the screen, don't add any functionality yet.
- If you want to add a 'feel' to the look, add some basic functionality, but don't overdo it, since it's only a prototype.
- Show it to all your friends.

5 An alternative way: the canvas

Not every user interface can be made using widgets. Take the desktop of Windows XX for example; it consists of a number of icons that have to be clicked to do something. Such an interface can not be constructed with the standard user interface widgets like buttons, listboxes, scrollbars, etc. The *canvas* can help us out. A canvas can be used to put *items* on a window. Items are standard elements of drawing: lines, rectangles, arcs, etc. If actions are bound to some events on these items, an alternative user interface can be made.

5.1 An example

```
# create and pack canvas:
canvas .c
pack .c
# create global images:
image create photo icon1 -file icon1.gif
image create photo icon2 -file icon2.gif
# create image items:
.c create image 50 50 -image icon1 -tags icon1tag
.c create image 100 50 -image icon2 -tags icon2tag
# bind some action to clicking on the items:
.c bind icon1tag <ButtonPress> { puts "click on icon 1" }
.c bind icon2tag <ButtonPress> { puts "click on icon 2" }
# draw some other stuff:
.c create arc 120 50 150 120 -fill green -tags greenthing
.c create line 40 160 160 160 -fill red
.c create oval 50 100 150 120 -outline yellow
.c create rectangle 20 20 180 180 -outline black
.c create text 20 185 -anchor nw -text "Canvas Example, please enjoy"
# bind an action to another item:
.c bind greenthing <Enter> { puts "cursor entered green-thing" }
.c bind greenthing <Leave> { puts "cursor left green-thing" }
```

First, the canvas is created and put onto the window. Some images are created next. This is done with the `image` command. The shown statements create two 'internal' images with names **icon1** and **icon2**. The image data is loaded from the given files. The names **icon1** and **icon2** can be used with the `-image` option of widgets. The `image` command can do a lot more, see [1] for more information.

The images are used to create a couple of image items on the canvas. The general command to create something on a canvas is
canvas-name **create** *type* ...
where type is one of **arc**, **bitmap**, **image**, **line**, **oval**, **polygon**, **rectangle**, **text** or **window**. See [1] for the options that this command can take with each type.

When an item is created on a canvas, a *tag-list* can be passed. Tags in this list can be used to refer to specific items. An example of this is the binding of some actions to the event clicking on the images. The tag is passed to the bind command determines to which item something will be bound.

The last two parts of the example show some other types of items and another binding.

5.2 The result

The resulting 'application' is shown in figure 9. Clicking **icon1** or **icon2** will result in a text printed on stdout. When the mouse cursor enters or leaves the arc (the item in the top right corner), again a text is printed.

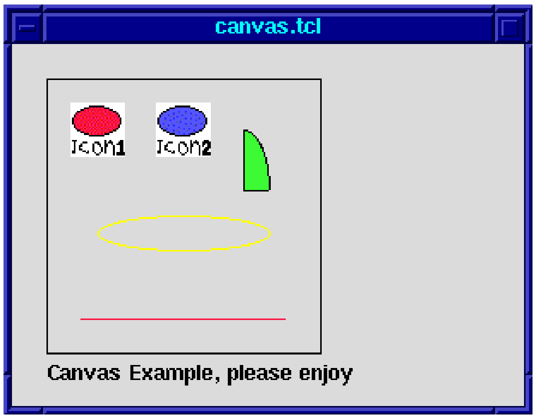

Figure 9: An example of a canvas with items

References

[1] Paul Raines and Jeff Tranter, *Tcl/Tk Reference Guide*, can be found in `/usr/local/doc/tcltk/tkref/refguide.ps`.

[2] SAM, *The Tk Packer*, can be found in `home/se/doc/packer/packer.ps`.