X Wrappers for Non-Graphic Interactive Programs

Don Libes

Factory Automation Systems Division
National Institute of Standards and Technology
Gaithersburg, MD 20899

Abstract

Expectk is a toolkit for wrapping character-oriented interactive programs in graphic user interfaces (GUIs). No changes to the programs themselves are made. This is convenient for the large number of existing tools in the UNIX toolbox such as passwd, rlogin, crypt, fsck, and tip, that are otherwise non-programmable and could take a large amount of time to make them so. Commercial software is often similarly limited, often comes without source, and hence no way to adapt to GUIs without complete rewriting of the application itself. Expectk can also be used to combine multiple programs together for synergistic effects. The underlying programs can be entirely hidden from view.

Keywords: Expect, Expect, Tk, GUI, Interactive Programs, Wrappers, Reuse, X Window System, X

To appear in: Proceedings of Xhibition 94, San Jose, California, June 20-24, 1994.

Introduction

Numerous legacy programs are interactive but non-graphic. In many cases, converting them to use graphic user interfaces (GUIs) would be beneficial. The traditional approach is to go back to the original source code, tear into it, rewriting as appropriate, and finally relinking. This is an expensive process. It is also likely to introduce bugs in parts of the program which were bug-free.

Older programs are more likely to be designed in a non-modular fashion. In such cases, replacing the user interface can be complex.

This paper describes Expectk, a software tool for providing existing programs with GUIs without any change to the underlying programs. Using Expect, it is possible to write GUIs that are simple and modular. It is much simper to write an Expectk-based GUI than to rewrite the existing application. Other benefits accrue as well.

Infrastructure

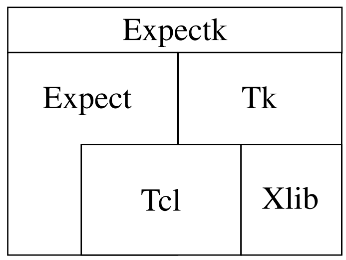
Expectk is the amalgamation of three other pieces of software: Tcl, Expect, and Tk.

The implementation and philosophy of Expect is described at length by Libes [5]. Briefly, scripts are written in an interpreted language. (A library is available for C and C++ programmers but it will not be further discussed in this paper.) Commands are provided to create interactive processes and to read and write their output and input. Expect is named after the specific command which waits for output from a program.

The language of Expect is based on Tcl. Tcl is actually a subroutine library, which becomes embedded into an application and provides language services. The resulting language looks very much like a typical shell language. There are commands to set variables (`set`), control flow (`if`, `for`, `continue`, etc.), and perform the usual math and string operations. Of course, UNIX programs can be called (`exec`). All of these facilities are available to any Tcl application. Tcl is completely described by Ousterhout [1][2].

Expect is built alongside of Tcl and provides additional commands. The `spawn` command invokes a UNIX program for interactive use. `send` sends strings to a process. `expect` waits for strings from a process. `expect` supports regular expressions and can wait for multiple strings at the same time, executing a different action for each string. `expect` also understands exceptional conditions such as time-out and end-of-file.

Tk is a library for creating Motif-like interfaces. Like Expect, Tk is also Tcl-based. Tk has commands such as `button` and `menu` to create graphic elements and `bind` to invoke actions (also specified in Tcl) when events occur. Tk is described further by Ousterhout [3][4].



Expectk is a combination of Tk and Expect which are in turn extensions to Tcl.

The Result – Expectk

Expectk is a program in which Expect and Tk can be used together. For example, the output of an interactive program can be made to appear in a scrolling window. The programmer is free to provide input in any way desired. Instead of having the user type it in, the programmer can offer the user a menu from which to select. Once chosen, input can be fed to the underlying programs. The Expect extension makes the programs think that it was typed in by a user.

Other graphic devices can be used as well such as button, sliders, and generic canvas for drawn figures. These graphics can be used for output as well as input.

The expect_background command

In Expect, commands are usually written to be executed sequentially. Programs in the X environment are more often event driven. To support this, an additional command was added besides those already in Expect. The `expect_background` command handles input from spawned processes whenever the Tk event loop is active (i.e., when Expectk is idling waiting for an event). Using `expect_background`, it is possible to have actions execute whenever input arrives and matches a pattern. Such patterns are called "background patterns".

The `expect_background` command takes arguments identically to the `expect` command. For example, the following command matches any input (the regular expression `.+` matches anything) coming from the process `$shell`. The pattern matcher stores the data in `$expect_out(0,string)`. This is then given to the text widget which is told to insert it at the end. The result is that everything from the shell appears in the text widget.

```
expect_background -i $shell -re ".+" {
    .text insert end $expect_out(0,string)
}
```

The **expect_background** command returns immediately. However, the patterns are remembered by Expect. Whenever any input arrives, it is compared against the pattern. If it matches, the action is executed.

The following commands start a shell and text widget for its output. Similarly, a **telnet** process is started with a text widget for it. The **expect_background** command allows future input from either a shell or a **telnet** process to go into their respective text widgets.

```
# start a shell and text widget for its output
spawn $env(SHELL)
set shell $spawn_id
text .shell
pack .shell

# start a telnet and a text widget for its output
spawn telnet
set telnet $spawn_id
text .telnet
pack .telnet

expect_background {
    -i $telnet -re ".+" {
        .telnet insert end $expect_out(0,string)
    } -i $shell -re ".+" {
        .shell insert end $expect_out(0,string)
    }
}
```

Why Is All Of This Good?

Existing Programs Are Reused

Expectk allows and encourages the reuse of existing programs. This avoids having to duplicate effort that has already been expended. For example, a GUI for **ftp** does not have to duplicate the effort invested internally to the **ftp** program such as socket allocation and management.

Reuse also permits Expectk scripts to avoid changes other than in the user-interface. For example, if changes occur in the password encryption algorithm or the location of the password database (i.e., local, shared, shadowed), there is no reason to change the user interface. Changes like these are appropriately made to underlying programs and are isolated there.

Hide Existing Programs

Underlying programs used by Expectk can be entirely hidden from the user. For example, there is little point to forcing the user to view a long dialogue showing the navigation through frontends.

Enhance Existing Programs

Existing programs can be enhanced by using Expectk. The **passwd** program, used to set user passwords, has no provision for verifying that passwords are not in the dictionary. Expectk can check this or other things before going ahead and calling upon **passwd**.

Ease Use of Existing Programs

Many existing programs are hard to use. They have user interfaces designed for computer scientists, UNIX experts, etc. It is possible to write front-ends for particular audiences to simplify use in most cases.

Combine Existing Programs

Existing programs can be combined using Expectk. For example, a chemical analysis expert system may lack the ability to evaluate arbitrary precision expressions – indeed, very few packages provide this. Using Expectk, the expressions can be fed through an arbitrary precision package such as **`bc`** or **`dc`** before being given to the expert system.

Less Errors Introduced

The traditional approach for building a GUI-version of an existing application is go into the working source code and tear out the user-interface. This can be extremely painful. Most line-oriented programs were not designed with a modular front-end. The result requires regression testing of application code that originally worked but is now suspect due to changes.

Using Expectk, no changes to the application are made. Therefore, no testing of the underlying application is necessary. The Expectk-based GUI is a wholly separate piece of software which runs as a separate process.

Fast to Write

Because programs do not have to be rewritten, the focus of Expectk script is entirely upon the user interface. This lowers the cost of producing a tool with a GUI.

Expectk is written using a high-level command language. It has direct support for Motif-like interfaces. It also has the direct ability to control existing interactive and non-interactive programs. Because the command-language is interpreted, there is no time-consuming recompilation phase. Scripts can be edited and rerun in seconds.

Scripts to pop up a couple of buttons or a menubar to control simple applications can be written in minutes. A automated GUI-builder [8] is available for very complicated layouts.

Fast to Execute

User interfaces rarely need high speed – although output can be demanding, user input is very slow (relatively speaking). While Expectk scripts are interpreted, close attention has been paid to speed in critical areas. However, the user interface is the only part that is interpreted. The underlying programs still run as before. The final result is typically faster than the original, since user inputs arrive as a single event (i.e, a button press) rather than typing "**`y`**", "**`e`**", "**`s`**", and <return>.

Example – tkpasswd

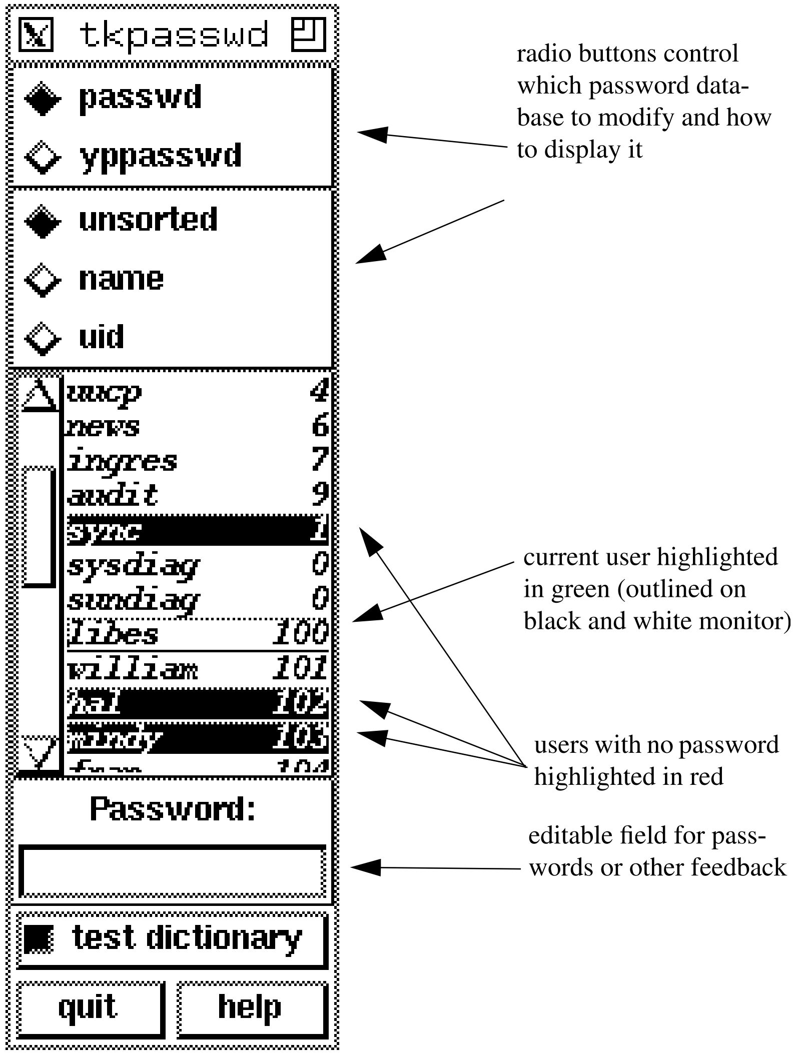
The **`tkpasswd`** program is a good example of how complexity can be avoided. While **`passwd`** is conceptually simple – even trivial – rewriting it is not. The **`passwd`** program must deal with things like shadow password files, password file locking, encryption algorithms, etc. By reusing **`passwd`**, the GUI wrapper inherits the correct behavior when it comes time to set the password. The GUI wrapper need only worry about the GUI and any additional functionality.

An extended example – tkpasswd

`tkpasswd` is an Expectk script that provides a system administrator with a GUI for conveniently changing passwords. The complete source to **`tkpasswd`** is about 300 lines. Only about 10 of the lines are directly related to driving the **`passwd`** program, but it is worthwhile to examine other parts of the program to see the interrelationships to how, for example, the press of a button is translated into a behavior change in the **`passwd`** interaction. The entire script comes with the Expectk distribution as an example.

When run, the script displays the following image. It is composed of several radiobuttons at the top which control the script. In the middle is a user browser. Below this are several more buttons and an entry widget in which passwords can be entered.

The first two buttons chooses between a local password database (**`/etc/passwd`**) or an NIS database. When either button is activated, the variable **`passwd_cmd`** is assigned the list of "**`passwd`**" and "**`cat /etc/passwd`**" or the list



"**yppasswd**" and "**ypcat passwd**". The first element in each list is the appropriate UNIX command to set a password and the second element is the appropriate UNIX command to read the password database.

Each time a button is pressed, the **get_user** command is also called to reload the correct password database.

Brief descriptions are provided for the graphical aspects but just for this first set of buttons. This will help to give you more of a feel for Tk if you've never used it. Both buttons are embedded in a frame with a raised border so that it is easy for the user to see that they are related. "**-bd 1**" means the border is one pixel wide. The "**-anchor w**" aligns the buttons on the west side of the frame. The pack command places the buttons into the frame and then the frame itself is placed on the screen. The "**-fill x**" makes the buttons and the frame expand horizontally to fill the display. (The actual width is determined later.)

```
frame .type -relief raised -bd 1
radiobutton .passwd -text passwd -variable passwd_cmd \
        -value {passwd {cat /etc/passwd}} \
        -anchor w -command get_users -relief flat
radiobutton .yppasswd -text yppasswd -variable passwd_cmd \
        -value {yppasswd {ypcat passwd}} \
        -anchor w -command get_users -relief flat
pack .passwd .yppasswd -in .type -fill x
pack .type -fill x
```

In another frame, three more buttons are created which control how users are sorted in the display. As before, a value is assigned to a control variable (**sort_cmd**) which conveniently is just the right UNIX command to sort a file. Providing sorted and unsorted displays is important as the NIS database is provided in a randomized order (which usually cries for sorting) while the local database is provided in the original order (and may or may not need sorting).

```
frame .sort -relief raised -bd 1
radiobutton .unsorted -text unsorted -variable sort_cmd \
       -value " " -anchor w -relief flat \
       -command get_users
radiobutton .name -text name -variable sort_cmd \
       -value "| sort" -anchor w -relief flat \
       -command get_users
radiobutton .uid -text uid -variable sort_cmd \
       -value "| sort -t: -n +2" \
       -anchor w -relief flat -command get_users
pack .unsorted .name .uid -in .sort -fill x
pack .sort -fill x
```

In the center of the display is a frame containing a user browser (user list and scrollbar). The users are displayed in a text widget. The currently selected user is displayed in green and users with no passwords are highlighted in red (to indicate danger). On a monochrome monitor, black on white and white on black is used with an additional border to distinguish between the usual white on black entries.

The default number of users shown is 10 but the window is defined so that the user can increase or decrease it, in which case the user list expands or contracts appropriately. The remainder of the display is fixed.

The width of the users is set at 14 – enough for an eight character user, a blank, and a 5 character user id. Everything else in the display is fixed to this width and the user is not allowed to change it.

```
frame .users -relief raised -bd 1
text .names -yscrollcommand ".scroll set" -width 14 \
       -height 1 -font "*-bold-o-normal-*-120-*-m-*" \
       -setgrid 1
.names tag configure nopassword -relief raised
.names tag configure selection -relief raised
if {[tk colormodel .]=="color"} {
    .names tag configure nopassword -background red
    .names tag configure selection -background green
} else {
    .names tag configure nopassword -background  black \
           -foreground white
    .names tag configure selection -background white \
           -foreground black
}
scrollbar .scroll -command ".names yview" -relief raised
pack .scroll -in .users -side left -fill y
pack .names  -in .users -side left -fill y
pack .users -expand 1 -fill y

wm minsize . 14 1
wm maxsize . 14 999
wm geometry . 14x10
```

A field within a frame labelled “`Password`” is provided in which the user can enter new passwords. The focus is moved to the entry field allowing the user to enter passwords no matter where the cursor is in the display. Special bindings are added (later) which allow scrolling via the keyboard as well.

```
frame .password_frame -relief raised -bd 1
entry .password -textvar password -relief sunken -width 1
focus .password
bind .password <Return> password_set
label .prompt -text "Password:" -bd 0
pack .prompt .password -in .password_frame -fill x \
       -padx 2 -pady 2
pack .password_frame -fill x
```

Several more buttons are created and placed at the bottom of the display. Rather than putting them at the top, they are placed at the bottom because it is likely they will be pressed at most once. In contrast, the buttons at the top are likely to be pressed many times.

The first button controls whether passwords are checked against a dictionary. It sets the variable **dict_check** appropriately, and the dictionary is loaded if it has not been already.

```
set dict_loaded 0
checkbutton .dict -text "test dictionary" \
        -variable dict_check \
        -command {
            if !$dict_loaded load_dict
        }
pack .dict -fill x -padx 2 -pady 2
```

A quit button causes the program to exit if pressed.

```
button .quit -text quit -command exit
button .help_button -text help -command help
pack .quit .help_button -side left -expand 1 \
        -fill x -padx 2 -pady 2
```

A help button pops up a help window describing how to use the program. The actual text is omitted here.

```
proc help {} {
    catch {destroy .help}
    toplevel .help
    message .help.text -text <...help text here...>

    button .help.ok -text "ok" -command {destroy .help}
    pack .help.text
    pack .help.ok -fill x -padx 2 -pady 2
}
```

It is interesting to note that all the preceding code is just to set up the display and takes about a third of the program.

The **get_users** procedure reloads the password database. It is called when any of the top buttons are activated.

After clearing the current list, the procedure executes the appropriate UNIX commands to read and sort the password database. The particular commands are defined by the radio buttons. They select which database to read and how to sort it.

The remainder of the procedure adds the users to the list of names, appropriately tagging any with null passwords. User ids are displayed as well. Usernames that have no other information with them are pointers back to the NIS database. They are displayed without user ids but nothing else is done. The script doesn't have to worry about them because the **passwd** program itself will reject attempts to set them.

```
proc get_users {} {
    global sort_cmd passwd_cmd
    global nopasswords;# line #s of users with no passwords
    global last_line;# last line of text box
    global selection_line

    .names delete 1.0 end

    set file [open "|[lindex $passwd_cmd 1] $sort_cmd"]
    set last_line 1
    set nopasswords {}
    while {[gets $file buf] != -1} {
        set buf [split $buf :]
        if [llength $buf]>2 {
            # normal password entry
            .names insert end "[format "%-8s %5d" [
                    lindex $buf 0] [lindex $buf 2]]\n"
            if 0==[string compare [lindex $buf 1] ""] {
```

```
                .names tag add nopassword \
                    {end - 1 line linestart} \
                    {end - 1 line lineend}
                lappend nopasswords $last_line
            }
        } else {
            # +name style entry
            .names insert end "$buf\n"
        }
        incr last_line
    }
    incr last_line -1
    close $file
    set selection_line 0
}
```

At various places in the script, feedback is generated to tell the user what is going on. For simplicity, feedback is displayed in the same field in which the password is entered. This is convenient as the user probably doesn't want the password left on the screen for long anyway. (Making the password entirely invisible could also be done.) The feedback is selected to make it appear more prominent to the user.

```
proc feedback {msg} {
    global password

    set password $msg
    .password select from 0
    .password select to end
    update
}
```

The dictionary takes considerable time to load into memory (about 10 seconds for a 25000 word dictionary on a Sun Sparc 2 workstation) so it is not loaded unless the user specifically activates the "**check dictionary**" button. The first time it is pressed, this procedure is executed. For each word, it creates an element in an array called "**dict**". No value is necessary. Later on, passwords will be looked up in the dictionary just by testing if the variable exists – a very fast operation.

Using the UNIX **grep** command would spread the load out, but it would also expose the password to anyone running **ps**. Instead, the procedure is sped up by careful Tcl coding.

```
proc load_dict {} {
    global dict dict_loaded

    feedback "loading dictionary..."

    if 0==[catch {open /usr/dict/words} file] {
        rename set s
        foreach w [split [read $file] "\n"] {s dict($w) ""}
        close $file
        rename s set
        set dict_loaded 1
        feedback "dictionary loaded"
    } else
        feedback "dictionary missing"
        .dict deselect
    }
}
```

The **weak_password** procedure is a hook in which you can put any security measures you like. As written, all it does is reject a word if it appears in the dictionary. The mechanism to look up a word was described earlier.

```
# put whatever security checks you like in here
proc weak_password {password} {
    global dict dict_check
```

```
    if $dict_check {
        feedback "checking password"

        if [info exists dict($password)] {
            feedback "sorry - in dictionary"
            return 1
        }
    }
    return 0
}
```

After entering a password, the **password_set** procedure is invoked to set the password. The interactive command is extracted from the radiobuttons and it is spawned. If the prompt is for an "**old password**", the script queries the user for it and then passes it on. The new password is sent as many times as requested without telling the user. (All passwords have to be entered twice. Short passwords have to be entered four times.) Any unrecognized response is passed back to the user.

```
proc password_set {} {
    global password passwd_cmd selection_line

    if {$selection_line==0} {
        feedback "select a user first"
        return
    }
    set user [lindex [.names get selection.first selection.last] 0]

    if [weak_password $password] return

    feedback "setting password . . ."

    set cmd [lindex $passwd_cmd 0]
    spawn -noecho $cmd $user
    log_user 0
    set last_msg "error in $cmd"
    while 1 {
        expect {
            -nocase "old password:" {
                exp_send "[get_old_password]\r"
            } "assword:" {
                exp_send "$password\r"
            } -re "(.*)\r\n" {
                set last_msg $expect_out(1,string)
            } eof break
        }
    }
    set status [wait]
    if [lindex $status 3]==0 {
        feedback "set successfully"
    } else {
        feedback $last_msg
    }
}
```

The script is intended to be run by a superuser. Traditionally, the superuser is not prompted for old passwords so no entry field is permanently dedicated in the display for this purpose. However, just in case the user is prompted, a window is popped up to collect the old password. This also handles the case when a non-superuser tries to change their own password. Trying to change any other password will be trapped by the password program itself so the script doesn't have to worry about it.

The procedure temporarily moves the focus to the popup so the user doesn't have to move the mouse. After pressing return, the popup goes away and the focus is restored.

```
proc get_old_password {} {
    global old

    toplevel .old
    label .old.label -text "Old password:"
    catch {unset old}
    entry .old.entry -textvar old -relief sunken -width 1

    pack .old.label
    pack .old.entry -fill x -padx 2 -pady 2

    bind .old <Return> {destroy .old}
    bind .old.entry <Return> {destroy .old}
    set oldfocus [focus]
    focus .old.entry
    tkwait visibility .old
    grab .old
    tkwait window .old
    focus $oldfocus
    return $old
}
```

Once enough procedures are defined, the script can initialize the user list and radio buttons. Initially, the local password database is selected and displayed without sorting.

```
.unsorted select
.passwd invoke
```

The remaining effort in the script is in handling user input. The global variable **selection_line** identifies the user whose password is about to be changed. The **make_selection** procedure scrolls the user list if the selected user is not displayed. Lastly the selected user is highlighted.

```
proc make_selection {} {
    global selection_line last_line

    .names tag remove selection 0.0 end

    # don't let selection go off top of screen
    if {$selection_line < 1} {
        set selection_line $last_line
    } elseif {$selection_line > $last_line} {
        set selection_line 1
    }
    .names yview -pickplace [expr $selection_line-1]
    .names tag add selection $selection_line.0 \
          [expr 1+$selection_line].0
}
```

The **select_next_nopassword** procedure searches through the list of users that do not have passwords. Upon finding one, it is highlighted. The procedure is long because it has to search in either direction and must start searching from the middle of the list and loop around if necessary.

```
proc select_next_nopassword {direction} {
    global selection_line last_line
    global nopasswords

    if 0==[llength $nopasswords] {
        feedback "no null passwords"
        return
    }
```

```
        if $direction==1 {
            # is there a better way to get last element of list?
            if $selection_line>=[lindex $nopasswords [expr [llength $nopasswords]-1]] {
                set selection_line 0
            }
            foreach i $nopasswords {
                if $selection_line<$i break
            }
        } else {
            if $selection_line<=[lindex $nopasswords 0] {
                set selection_line $last_line
            }
            set j [expr [llength $nopasswords]-1]
            for {} {$j>=0} {incr j -1} {
                set i [lindex $nopasswords $j]
                if $selection_line>$i break
            }
        }
        set selection_line $i
        make_selection
    }
```

The `select` procedure is called to figure out which user has been clicked on with the mouse. Once it has, it updates `selection_line` and the display.

```
    proc select {w coords} {
        global selection_line

        $w mark set insert "@$coords linestart"
        $w mark set anchor insert
        set first [$w index "anchor linestart"]
        set last [$w index "insert lineend + 1c"]
        scan $first %d selection_line

        $w tag remove selection 0.0 end
        $w tag add selection $first $last
    }
```

The bindings are straightforward. Mouse button one selects a user. ^C causes the application to exit. ^N and ^P moves the user up and down by one. M-n and M-p invokes `select_next_nopassword` to find the next user without a password. These bindings are defined for the entry field in which the new password is entered. Because this field always has the focus, the user can select different users and enter passwords without touching the mouse.

```
    bind Text <1> {select %W %x,%y}

    bind Entry <Control-c>{exit}

    bind .password <Control-n> \
        {incr selection_line 1;make_selection}
    bind .password <Control-p> \
        {incr selection_line -1; make_selection}
    bind .password <Meta-n>{select_next_nopassword 1}
    bind .password <Meta-p>{select_next_nopassword -1}
```

Using The Expectk Framework for Developing New Non-GUI Programs

Most of this paper focuses on the ability to use already existing non-GUI programs as a base for development of GUI counterparts. It is also possible to use this structure in developing new GUI software which does not use existing programs. To restate in other terms, GUI software can often be decomposed into a functional piece and a graphical piece (and maybe others).

The non-graphic functionality of a program often need not be imbedded in a GUI. By using a separate executable with a simple command interface, the program can be used directly on a dumb terminal, or it can be combined in the traditional UNIX fashion with other programs, or it may be used by Expectk, and so on. Related benefits include simpler unit testing, unit replacement, etc.

Conclusion

This paper has described techniques to take existing non-graphic programs and wrap them in X GUIs. In addition, these techniques allow multiple programs to work together whether or not they were originally intended or designed to do so. We believe this to be an efficient and economical approach for bringing existing applications into the new world of graphic user interfaces.

Availability

Since the design and implementation of this software was paid for by the U.S. government, it is in the public domain. However, the author and NIST would appreciate credit if this software, documentation, ideas, or portions of them are used.

The tools described in this document may be **`ftp`**'d as **`pub/expect/expect.tar.Z`**[1] from **`ftp.cme.nist.gov`**. The software will be mailed to you if you send the mail message "**`send pub/expect/expect.tar.Z`**" (without quotes) to **`library@cme.nist.gov`**.

Acknowledgments

Thanks to Bennett Todd, Craig Schlenoff, Przemek Klosowski, and Jan Putman for providing suggestions that greatly enhanced the readability of this paper.

The author gratefully acknowledges John Ousterhout for creating Tcl. Not only does Tcl solve a significant problem in software design, but the code itself as well as the documentation are comprehensive and written with consummate style. Tcl is truly a pleasure to use.

Portions of this work were funded by the NIST Scientific and Technical Research Services as part of the ARPA Persistent Object Base project, and the Continuous Acquisition and Life-Cycle Support (CALS) program of the Office of the Defense CALS Executive (ODCE).

Disclaimers

Trade names and company products are mentioned in the text in order to adequately specify experimental procedures and equipment used. In no case does such identification imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the products are necessarily the best available for the purpose.

References

[1] Ousterhout, John, "Tcl: An Embeddable Command Language", *Proceedings of the Winter 1990 USENIX Conference*, Washington, D.C., January 22-26, 1990.

[2] Ousterhout, John, "Tcl(3) – Overview of Tool Command Language Facilities", *unpublished manual page*, University of California at Berkeley, January 1990.

[3] Ousterhout, John, "Tcl and the Tk Toolkit", Addison-Wesley, April 1994.

1. The ".Z" file is compressed. A ".gz" version is also available which is gzipped.

[4] Ousterhout, John, “An X11 Toolkit Based on the Tcl Language”, *Proceedings of the 1991 Winter USENIX Conference*, pp 105-115, 1991.

[5] Libes, Don, “Expect: Curing Those Uncontrollable Fits of Interaction”, *Proceedings of the Summer 1990 USENIX Conference*, pp. 183-192, Anaheim, CA, June 11-15, 1990.

[6] Libes, Don, “Expect: Scripts for Controlling Interactive Programs”, *Computing Systems*, pp. 99-126, Vol. 4, No. 2, University of California Press Journals, CA, Spring 1991.

[7] Libes, Don, “Expectk”, *unpublished manual page*, National Institute of Standards and Technology, January 1993.

[8] Delmas, Sven, “XF”, *unpublished manual page*, TU Berlin, Germany, 1993.