Tcl Programming Techniques for Specifying Visual Interfaces

Jeffrey P. Lankford*
Automation Sciences Laboratory
Northrop Grumman Corporation
Pico Rivera, CA 90660
`JLankford@NRTC.Northrop.COM`

April 1, 1994

Using Tcl source code, this paper reviews three common programming approaches: procedure-oriented, data-oriented, and object-oriented. A single simple application, specifying a pull-down cascade menu comprised of pushbutton entries, illustrates these approaches. Though Tcl core is not object oriented, we show a technique for integrating Tcl and an object-oriented visual specification language using Tcl's C language embedding capability.

Procedure-Oriented

The first approach is the most direct. A programmer sequentially enters instructions to generate visual components using some application programming interface (API). Figure 1 shows the example using tclMotif.[1]

This approach is tedious and error-prone, because of repetitive but slightly varying code. For example, altering a single parameter, say "ctrl+" to "Ctrl<Key>", requires changing many lines of code. Hence, this verbose code is difficult to maintain.

Data-Oriented

The second approach uses "loop-rolling", by which programmers separate invariant code from varying parameters. Figure 2 shows parameters grouped in a table and invariant code moved to loop bodies. Variant code parameters moderate invariant code behavior.

This approach does simplify program maintenance. New visual objects are introduced simply by adding them to the data set. Increased behavior control is achieved by extending the set of parameter slots and recoding procedures as appropriate. Component parameter value definitions are isolated from component behavior changes. For example, the procedure `acToAcText` can be enhanced incrementally without changing parameter declarations.

Object-Oriented

The third approach uses modular object definition to both encapsulate and separate parameter and method specification. By inheriting definitions for object parameters and methods, code comprising large programs is reduced.

Figure 3 shows object-oriented specification using OSF/Motif UIL[3]. Though the advantages of the technique are not fully demonstrated (partly because the example is too brief and partly because of excessive lexical ornamentation of the language), inheritance of the `XmNborderWidth` argument suggests improvements obtainable in larger programs. OSF/Motif UIL is used, since no comparable descriptive visual specification language is available with Tcl.

*The author appreciates contributions by Automation Sciences Laboratory staff preparing this paper for publication.

[1]Two-column layout necessitates peculiar code format. The notation "..." indicates excised code.

One difficulty arises attempting to use UIL for visual component specification and Tcl for application behavior specification: UIL assumes run-time binding routines that link callback events to C, rather than Tcl, routines. Figure 4 illustrates the flexibility of Tcl's C language embedding feature in circumventing the restriction to C code definitions of callback methods.[2]

The first fragment shows a custom language[3] definition of application callback routines.[4] In practice, Tcl code is substituted for the comment. The second fragment shows the declaration after conversion to Tcl code, with the procedure body passed through unaltered. The third fragment shows the definition after it has been compiled into C code. The body of the procedure is a parameterized stub that dynamically forms and executes a Tcl command. The C stub collects callback arguments with the Tcl procedure name, and converts arguments to strings as necessary. It calls a previously initialized Tcl interpreter to evaluate the command in the global context, as is ordinarily done for direct Tcl callbacks.

Past and Future

OSF/Motif is the API standard used by the Automation Sciences Laboratory for programming visual interfaces. We considered coercing Tk to emulate both Motif's visual and programming interface, but deemed it too costly. Newmarch's[2] tclMotif extension proved invaluable. One benefit of tclMotif is that it uses the same API for both Tcl and C, offering a natural optimization path.

Descriptive visual specification languages facilitate rapid prototyping of visual interfaces. Had OSF/Motif UIL not been available, a comparable visual specification language would likely have been developed. Using an existing language and compiler saved effort.

[2]This example derives from Ousterhout[4].

[3]The parser for this language was developed using the m4 macro processor.

[4]Use of the special Tcl formal parameter `args` is merely illustrative, not significant.

None of the examples presented illustrate use of McLennan's[1] [incr tcl] extension. [incr tcl] supports the richer object-oriented data structuring useful for more elegant interfacing of C and Tcl callback routines. Similar to tclMotif, [incr tcl] offers a convenient path from Tcl to C++ programming.

A distributed software engineering management environment developed at Northrop uses a suite of visual interfaces to view databases and manipulate tools. Applications based on a common visual interface framework include an engineering process definition editor, a process life-cycle simulation monitor, and a process operation controller. Using the object-oriented approach to separate specification of visual components from application behavior facilitates concurrent development of applications and their visual interfaces.

References

[1] M.J. McLennan. [incr tcl] – Object-Oriented Programming in Tcl. *Tcl/Tk Workshop Proceedings*, U.C. Berkeley, June 1993.

[2] J. Newmarch. A Tcl Binding to Motif. University of Canberra, available via Internet `anonymous` FTP from `harbor.ecn.purdue.edu` in the `/pub/motif/` directory.

[3] Open Software Foundation. *OSF/Motif Programmer's Guide*, Revision 1.1. Prentice Hall, Englewood Cliffs, NJ, 1991.

[4] J.K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, MA, 1993.

Jeff Lankford received the M.S. degree in Engineering Science from the University of Tennessee Space Institute in 1983. Current research interests include visual user interfaces for process-centered software engineering modeling, simulation, and operations control. He is a member of ACM and IEEE.

```
        ...
set toplevel ".main"
        ...
xmPulldownMenu $toplevel.filePane
xmCascadeButton $toplevel.menuBar.File \
  managed -subMenuId $toplevel.filePane
$toplevel.menuBar.File setValues \
  -borderWidth 0 -mnemonic "F"
xmPushButton $toplevel.filePane.New managed
$toplevel.filePane.New activateCallback \
  newFileCB
$toplevel.filePane.New setValues \
  -borderWidth 0 -mnemonic "N" \
  -accelerator "^N" -acceleratorText "Cntl+N"
xmPushButton $toplevel.filePane.Open managed
$toplevel.filePane.Open activateCallback \
  openFileCB
$toplevel.filePane.Open setValues -mnemonic
  "O" -labelString "Open..." -borderWidth 0 \
  -accelerator "^O" -acceleratorText "Cntl+O"
xmSeparator $toplevel.filePane.line
xmPushButton $toplevel.filePane.Save managed
$toplevel.filePane.Save activateCallback \
  saveFile
$toplevel.filePane.Save setValues \
  -borderWidth 0 -mnemonic "S" \
  -accelerator "" -acceleratorText "Cntl+S"
xmPushButton $toplevel.filePane.SaveAs managed
$toplevel.filePane.SaveAs activateCallback \
  saveAsFileCB
$toplevel.filePane.SaveAs setValues -mnemonic\
  "A" -borderWidth 0 -labelString "Save As..."
xmPushButton $toplevel.filePane.Print managed
$toplevel.filePane.Print activateCallback \
  printFileCB
$toplevel.filePane.Print setValues -mnemonic\
  "P" -labelString "Print..." -borderWidth 0\
  -accelerator "^P" -acceleratorText "Cntl+P"
xmPushButton $toplevel.filePane.quickPrint \
  managed
$toplevel.filePane.quickPrint activateCallback\
  quickPrintFileCB
$toplevel.filePane.quickPrint setValues \
  -mnemonic "Q" -labelString "Quick Print" \
  -borderWidth 0 -accelerator "" \
  -acceleratorText "Cntl+Q"
xmPushButton $toplevel.filePane.Exit managed
$toplevel.filePane.Exit activateCallback \
  exitFileCB
$toplevel.filePane.Exit setValues \
  -mnemonic "E" -borderWidth 0
        ...
```

Figure 1: Tcl procedure-oriented example.

```
        ...
set filePaneEntrys { \
{.New "New" "N" "Ctrl<Key>N" newFileCB} \
{.Open "Open..." "O" "Ctrl<Key>O" \
 openFileCB} \
{} \
{.Save "Save" "S" "Ctrl<Key>S" saveFileCB}\
{.SaveAs "Save As..." "A" "" saveAsFileCB}\
{} \
{.Print "Print..." "P" "Ctrl<Key>P" \
 printFileCB} \
{.quickPrint "Quick Print" "Q" "Ctrl<Key>Q" \
 quickPrintFileCB} \
{} \
{.Exit "Exit" "E" "" exitFileCB} \
}
        ...
createCascadeMenu .main.filePane $filePaneEntrys
        ...
proc createCascadeMenu {menu entrys} {
  foreach a $entrys {
    if {$a == {}} {
      xmSeparator $menu.line managed
    } else {
      if {[lindex $a 5] == {}} {
        createCMEntry $menu[lindex $a 0] \
         [lindex $a 1] [lindex $a 2] \
         [lindex $a 3] [lindex $a 4]
      } else {
        createCMEntry $menu[lindex $a 0] \
         [lindex $a 1] [lindex $a 2] \
         [lindex $a 3] \
         [list [lindex $a 4] [lindex $a 5] ]
      } } } }
proc createCMEntry {button label mnemonic \
 accelerator callback} {
  xmPushButton $button managed
  if {$accelerator == ""} {
    $button setValues -borderWidth 0 \
     -labelString $label -mnemonic $mnemonic
  } else {
    $button setValues -borderWidth 0 \
     -labelString $label -mnemonic $mnemonic \
     -accelerator $accelerator \
     -acceleratorText [acToAcText $accelerator]
  }
  $button activateCallback $callback
}
proc acToAcText {string} {
  return $string
}
        ...
```

Figure 2: Tcl data-oriented example.

```
        ...
procedure
  newFileCB        ();
  openFileCB       ();
        ...
  exitFileCB       ();
        ...
value
  k_new_label_text         : "New";
  k_open_label_text        : "Open...";
        ...
  k_exit_label_text        : "Exit";
        ...
object
  file_menu  : XmPulldownMenu {
    arguments {
      XmNlabelString="File";
      XmNborderWidth=0; };
    controls {
      XmPushButton {arguments {
        XmNlabelString=k_new_label_text;
        XmNmnemonic="N";
        XmNaccelerator="^N";
        XmNacceleratorText="Ctrl<key>N"; };
                    callbacks {
        MrmNactivateCallback=procedure
         newFileCB (); }; };
      XmPushButton {arguments {
        XmNlabelString=k_open_label_text;
        XmNmnemonic="O";
        XmNaccelerator="^O";
        XmNacceleratorText="Ctrl<key>O"; };
                    callbacks {
        MrmNactivateCallback=procedure
         openFileCB (); }; };
      XmSeparator  {arguments { }; };
        ...
      XmSeparator  {arguments { }; };
      XmPushButton {arguments {
        XmNlabelString=k_exit_label_text;
        XmNmnemonic="E"; };
                    callbacks {
        MrmNactivateCallback=procedure
         exitFileCB (); }; }; };
    callbacks {
      MrmNcreateCallback=procedure
       create_proc (k_file_menu); }; };
        ...
```

Figure 3: Motif UIL object-oriented example.

```
        ...
CALLBACK(`newFileCB',
  ARGLIST(
    ARG(w,`Widget',),
    ARG(t,`int *',),
    ARG(args,`XmListCallbackStruct *',)),
  `{
  ## Tcl code for callback action
}')
        ...
proc newFileCB {w t args} {
  ## Tcl code for callback action
}
        ...
static void  newFileCB
(Widget w, int * t, XmListCallbackStruct * args)
{
    extern Tcl_Interp*  interp;
    static char script[1024];

    data2string (script, "%s %W %d %L\n",
      "newFileCB", w, t, l);
    code = Tcl_GlobalEval(interp, script);
    if (*interp->result != 0)   {
        printf("%s\n", interp->result);
    }
}
        ...
```

Figure 4: m4, Tcl, and C callback example.