TASH: A Free Platform-Independent Graphical User Interface Development Toolkit for Ada

Terry J. Westley
Calspan SRL Corporation
twestley@acm.org

Abstract

A platform-independent Application Programming Interface (API) for developing Graphical User Interfaces (GUI) is described. This API includes a complete "thin" binding to Tcl and an experimental "thick" binding to Tk from Ada 95. Several features of Ada 95 such as access to subprograms, tagged types, and interface to C were used in this binding.

What is a PIGUI?

A Platform-Independent Graphical User Interface (PIGUI) toolkit is a

> software library that a programmer uses to produce GUI code for multiple computer systems. The toolkit presents functions and/or objects (along with a programming approach) which is independent of which GUI the programmer is targeting... Native look-and-feel is a desirable feature, but is not essential for PIGUIs. [4]

Examples of PIGUIs

Some PIGUIs currently available for Ada are Screen Machine from Objective Interface Systems (info@ois.com) and OpenUI from Open Software Associates (http://www.osa.com.au).

Perhaps the "hottest" PIGUI being discussed today among Internet programmers is the Java AWT (Abstract Window Toolkit) from Sun Microsystems (http://java.sun.com).

Introduction to TASH

TASH [8] is a binding to Tcl/Tk [6] from Ada (requires Ada 95). This binding allows a Tcl/Tk program to use Ada in place of C to implement Tcl command procedures. This may be done to increase execution speed, use complex data structures not available in Tcl, and to otherwise use capabilities of Ada not found in Tcl. It also makes the Tcl/Tk library functions available to an Ada program. These include string and list handling, regular expression matching, hash tables, and (most important for this discussion) access to a graphical user interface toolkit.

The binding to Tcl is a complete, thin[1] binding to Tcl version 7.5a2. Work is progressing on upgrading it to version 7.5. Thicker binding support is provided by using Ada data types and exceptions for error handling in overloaded subprograms.

The binding to Tk is an experimental thick binding to Tk version 4.1a2. This paper describes a sample Tk program and discuses some of the rationale for decisions made in designing this binding.

Introduction to Tcl/Tk

Tcl/Tk is [8]:

Free: the source code is copyrighted by the Regents of the University of California and Sun Microsystems and is freely available (via anonymous ftp) with few restrictions,

Interpreted: although experimental compilers are available (and the Tcl/Tk development group at Sun plans to build one), the code is more commonly interpreted,

General purpose: provides general programming facilities such as variables, iteration, condition evaluation, and procedures,

Extensible: the language can be easily extended in Tcl as well as in C. One of the most popular extensions is Tk, a toolkit for building graphical user interfaces based on the X Window System, Microsoft Windows, and Macintosh,

Embeddable: its interpreter is implemented as a library of C functions which can be incorporated into and extended by a larger application.

Portable: beginning with Tcl7.5 and Tk4.1, the core Tcl/Tk system is available for Macs and PCs as well as for Unix.

Some of the advantages of using Tcl in an application are:

Rapid development: Because of its extensibility and availability of add-on packages, Tcl, along with Tk, provide a higher level language than C. Being an interpreted language also results in quicker turn-around than the traditional edit-compile-link-test cycle of software development.

System integration: Tcl makes an excellent language for integrating several programs into one single user environment. Because it is embeddable, Tcl can also serve as an inter-application language. Many also hope it to be used in the future as an internetworking software agent language.

---

[1]A "thin" binding attempts to faithfully provide a one-to-one mapping to each data type, function call, and object in the target system.

User programming: Where appropriate, users can program an application directly in the scripting language. Many very successful applications, such as AutoCad and Excel, provide user-level programming through a custom language. With Tcl, the application designer can provide for user programming without having to design and implement a new language.

Platforms

With introduction of Tcl 7.5 and Tk 4.1, Tcl/Tk is available for Macintosh and PC platforms in addition to Unix [7]. Source code can be downloaded for many platforms. Binary programs and libraries are also available for Macs and PCs.

The TASH binding has been built with the freely available GNU New York University Ada Translator (GNAT) [5]. This system is available for several Unix variants and PCs running Windows 3.1 and Windows 95. It is also available for the MachTen operating system on Macs.

TASH itself is distributed as source code so that it may be built in any system that supports both Tcl/Tk and GNAT. See Appendix A for instructions for downloading and building TASH.

Sample Tcl/Tk program

Figure 1 shows the window created when a sample Tcl/Tk program, **`timer`**, is executed. This program was adapted from the timer program included with the Tk distribution. It implements a simple stop watch which has the ability to start, stop, and reset a timer.

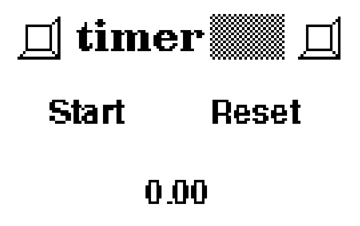

Figure 1

Figures 2 through 8 contain the source code of the timer program. The complete Tcl program is obtained by concatenating the code in these figures. This program is available in its entirety via anonymous ftp. See Appendx A for instructions for downloading it.

The Tcl version of the timer program is explained in this section and is contrasted with the Ada version in a later section. It is comprised of five procedure declarations and several other statements.

Figure 2 has nothing to do with the Tk binding. It shows a Unix technique for making a script executable and is required if we want to use this Tcl script as a stand-alone executable program. The permissions on the file containing this script must be set to executable for this technique to work.

Figures 3 through 7 contain the declaration of five procedures, **`Update`**, **`tick`**, **`Start`**, **`Reset`**, and **`Stop`**. These define new Tcl command procedures which extend the Tcl language to implement the primitives of this application.

```
#!/usr/local/bin/wish -f
```

Figure 2

Figure 3 declares a procedure which updates the timer window with the current value of the timer. When the timer script is executed, several subwindows are created within the main Tk window (these are shown in Figure 1). One of these windows, the **`.counter`** window, displays the current value of the timer. The **`Update`** procedure writes the current value of the timer into this window.

```
# Update the window by displaying the
# current value of the timer.
proc Update {} {
   global seconds hundredths
   .counter config -text [format \
      "%d.%02d" $seconds $hundredths]
}
```

Figure 3

The **`tick`** procedure is shown in Figure 4. This procedure increments the timer's value and displays it. It uses the **`after`** command to schedule itself to run again in the future to keep the timer running. The timer is stopped by setting the global variable, **`stopped`**, to 1 (Tcl value for True). When the tick procedure finds the **`stopped`** variable to be true, it does not schedule itself to run again.

```
# Increment the timer by one “tick.”
# A tick is 50 milliseconds (or 5 hundredths
# of a second).
proc tick {} {
   global seconds hundredths stopped
   # If the timer is stopped, do not
   # increment its value.
   if $stopped return
   # Schedule tick to be called again in
   # 50 milliseconds.
   after 50 tick
   # Increment the timer value.
   set hundredths [expr $hundredths+5]
   if {$hundredths >= 100} {
      set hundredths 0
      set seconds [expr $seconds+1]
   }
   # Update the timer display
   Update
}
```

Figure 4

The timer is started running by the **`Start`** procedure shown in Figure 5. It is only started if it is currently stopped. This prevents **`tick`** from being called again and starting a duplicate set of timer increments. Nothing is assumed about the value of the timer. It could have been reset to zero by the user, but this is not necessary for proper operation of the timer. If it was not reset to zero, it continues incrementing from the last value.

The **`Reset`** button shown in Figure 1 also serves as a **`Stop`** button. While the timer is stopped, it is labeled **`Reset`**. When the **`Start`** button is pressed, the **`Start`** procedure relabels the **`Reset`** button to be a **`Stop`** button.

```
# Start the timer if it is currently
stopped.
# Also, change the Stop button (currently
# labeled "Reset") to display "Stop."
proc Start {} {
   global stopped
   if $stopped {
      set stopped 0
      .stop config -text Stop -command Stop
      tick
   }
}
```

Figure 5

Figure 6 shows the **Reset** procedure which simply sets the value of the timer to 0.0 seconds and updates the display. It also initializes the global variable, **stopped**, so that it is in the correct state when the **Start** button is pressed, invoking the **Start** procedure.

```
# Reset the timer's value to 0.0 and update
# the display.
proc Reset {} {
   global seconds hundredths stopped
   set seconds 0
   set hundredths 0
   set stopped 1
   Update
}
```

Figure 6

The **Stop** procedure of Figure 7 is responsible for setting the global variable, **stopped**, to stop the timer. Once set to 1, this variable will cause the **tick** procedure to not reschedule itself to increment the timer value.

Once the timer is stopped, we have no need for a **Stop** button. So, it is relabeled and its command is modified to function as the **Reset** button.

```
# Stop incrementing the timer.  Also,
relabel
# the Stop button to be a Reset button.
proc Stop {} {
   global stopped
   set stopped 1
   .stop config -text Reset -command Reset
}
```

Figure 7

The first code which executes in the timer script is shown in Figure 8. There are three widgets[2] created and displayed by this program. This code creates and maps these widgets to the window.

The **label** command creates the text window in which the current timer value is displayed. The name of the label widget is **.counter**. It is initialized to the string "0.00" with a raised relief appearance.

The **pack** command causes the specified widget to be mapped onto the display in a particular orientation with respect to the parent window and previously mapped widgets. In this example, the label widget is the first to be mapped and is positioned at the bottom of the parent window.

```
label .counter -text 0.00 -relief raised \
   -width 10
pack .counter -side bottom -fill both

button .start -text Start -command Start
pack .start -side left -fill both \
   -expand yes

button .stop -text Reset -command Reset
pack .stop -side left -fill both \
   -expand yes

bind . <Control-c> {destroy .;exit}
bind . <Control-q> {destroy .;exit}

Reset
```

Figure 8

The two remaining widgets are buttons created to start, stop, and reset the timer. The Start button, created by the **button .start** command, executes the **Start** procedure whenever it is activated (by clicking the mouse over the button). If the timer is currently stopped, the **Start** procedure reprograms the other button to be a Stop button, then starts the timer.

The **Stop** button is created by the **button .stop** command. This button serves also as a reset button. When it is first created, it appears and acts as a **Reset** button. Later, it is converted to a **Stop** button in the **Start** procedure. When activated, this button calls either the **Reset** or **Stop** procedure depending on the current state of the timer.

The two **bind** commands associate handlers with key press events. When the key is pressed, the handler destroys the window and exits the program.

Reset is a call to the **Reset** procedure. It merely resets the counter and updates the timer value on the screen.

When the **wish** program reading this script encounters the end of file, it enters the Tk main event loop. Thereafter, all processing is initiated by event handlers until the program exits.

Time keeping is accomplished in the **tick** procedure by updating the **seconds** and **hundredths** counters, updating the label widget, and rescheduling the **tick** procedure to run again in 50 milliseconds.

This example gives a glimpse of the form and syntax of Tcl code and a flavor for the event handling nature of Tk. It also demonstrates how much can be accomplished in building a GUI application with very little code. This full application requires fewer lines of code than some graphical "hello world" programs.[3]

[2]The X Window System uses the term "widget" to denote a displayed component which can be independently created and controlled. The user interface is made up of many widgets.

[3]See the Louisiana Tech ACM "Hello World" project at http://www.latech.edu/~acm/HelloWorld.shtml. Note especially the Visual C++ entry.

An Ada Version of Timer

Figures 10 through 20 contain the source code of the Ada version of the timer program. When the code in each of these figures is concatenated, it makes up the complete Ada program. It's function and purpose is identical to that of the Tcl/Tk version shown in figures 2 through 8. Its appearance on the screen is indistinguishable from the Tcl/Tk version (Figure 1).

Before examining the Ada code in detail, let us first consider the overall pattern of a program which uses the Tcl/Tk library. Figure 9 is adapted from Figure 28.1 of [6].

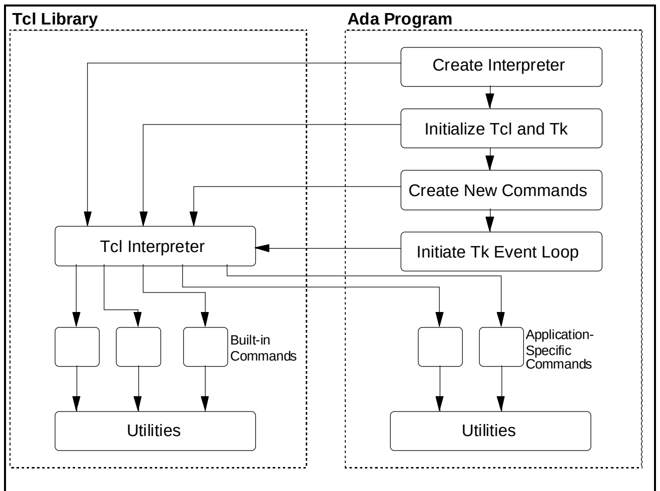

Figure 9

The Tcl Library consists of a Tcl script code interpreter and many built-in commands and utility procedures. The Tcl distribution includes procedures implemented in C as well as many implemented in Tcl.

A programmer may extend the Tcl script language by implementing new Tcl commands in C or in Tcl script code. Figures 3-7 show examples of procedures written in Tcl script code. These procedures are called, parsed, and executed by the Tcl interpreter exactly as any built-in command.

Application-specific commands may be implemented in C by adhering to a specific function prototype and return code convention. A new command is then created by registering the function with the Tcl intepreter.

Clearly, there are many advantages to implementing a new Tcl command in C. These include increased speed of execution and the capability to use complex data structures not available in Tcl.

In addition to the components of the Tcl Library, Figure 9 shows those of the application program. It first creates a Tcl interpreter, initializes Tcl and Tk, declares and registers new Tcl commands, then calls the Tk main event loop. The remainder of this section focuses on these components and how they can be implemented in Ada with TASH.

Figure 10 shows code required for the Ada version which has no equivalent function in the Tcl/Tk version. This code is not only necessary in the Ada version, but serves the purpose of readability and reliability for which Ada is well known.

For example, there is no need for **with** context clauses in Tcl. Consequently, the Tcl programmer has a more difficult time than the Ada programmer identifying which packages are used in a given program and in which package a given procedure is declared.

```
with C_Aux;
with Interfaces.C.Strings;
with Text_IO;
with Tcl;
with Tk;

procedure Timer is -- Timer

   package C renames Interfaces.C;

   package CreateCommands is new
      Tcl.Generic_CreateCommands (Integer);
   package Int_IO is new
      Text_IO.Integer_IO (Integer);

   Interp       : Tcl.Interp_Ptr;
   Result       : C.Int;
   Counter      : Tk.Label;
   Start_Button : Tk.Button;
   Stop_Button  : Tk.Button;
   Seconds      : Natural := 0;
   Hundredths   : Natural := 0;
   Stopped      : Boolean := True;
```

Figure 10

The instantiation of the generic package **Tcl.Generic_CreateCommands** provides the ability to create Tcl commands in Ada. Each of the created commands allows for the introduction of client data upon which the command can operate. In the timer example, no client data is needed, but since a client data type is a required part of the interface, **Integer** is used as a place holder for the client data.

Several variables are also declared. These include access to the Tcl interpreter, a result code for calling Tcl functions, screen widgets, and the **Seconds**, **Hundredths**, and **Stopped** variables.

The use of strong typing is credited with improving the reliability and correctness of a program since many program defects are discovered at compile-time rather than run-time.

Strong type may not be so critical in such a small application as this timer program. But, it does illustrate the benefits of using Ada variables over the use of Tcl variables to attain the strong typing benefits. A much larger application coded in Ada would greatly benefit from avoiding easy and innocent misuse of Tcl variables. All Tcl variables are strings and so can be easily misused.

Figures 11 through 17 contain the declaration of procedures which correspond to the Tcl procedures shown in Figures 3 through 7. The **Update** procedure of Figure 11 updates the **Counter** widget (a **Label** widget) with the current value of the timer. Notice that the reliability of the Ada version is higher than the Tcl/Tk because the **Seconds** and **Hundredths** variables are typed and cannot contain non-integer data.

```
   -- Update the window by displaying the
   -- current value of the timer.
   procedure Update is
      Hundredths_Image : String (1..2);
   begin -- Update
      Int_IO.Put (Hundredths_Image,
Hundredths);
      Tk.Configure (Counter, "-text " &
        Natural'image (seconds) & "." &
        Hundredths_Image);
   end Update;
```

Figure 11

The **Tick** procedure shown in Figure 12 increments the timer, then displays its value by calling the **Update** procedure. We chose to use Ada variables for **Seconds**, **Hundredths**, and **Stopped** because there was no advantage to using Tcl variables and there are significant reliability and efficiency advantages to using Ada variables.

```
   -- Increment the timer by one "tick."
   -- A tick is 50 milliseconds (or 5
hundredths
   -- of a second).
   procedure Tick is
   begin -- Tick
      -- if the timer is stopped, do not
      -- increment its value or reschedule
      -- tick for future execution.
      if Stopped then
         return;
      end if;
      -- Schedule tick to be called again in
      -- 50 milliseconds.
      Tk.After (50, "tick");
      -- Increment the timer value
      Hundredths := Hundredths + 5;
      if Hundredths >= 100 then
         Hundredths := 0;
        Seconds := Seconds + 1;
      end if;
      -- Update the timer display.
      Update;
   end Tick;
```

Figure 12

Figure 13 shows the **Tick** procedure wrapped with the necessary interface to enable it to be a Tcl procedure. This includes the specification of the function (to match a subprogram access type) as well as the rules for processing and return codes. The instantiated generic, **CreateCommands**, will be used to register the function as a procedure callable from Tcl.

The actual **Tick** procedure is factored out of the **Tick** command procedure so that it can be called directly from Ada code, not just as a Tcl procedure. The **Tick** procedure is encapsulated in this way into a Tcl command procedure so that it can be called from the **After** event scheduler. This procedure is an example of the Utilities routines shown in the Ada Program box of Figure 9.

An alternative design could use an Ada task with a delay statement to perform the clock ticks. Since X is not reentrant, however, some method of sequentializing calls to Tk would have to be implemented as well.

```
   -- Declare a procedure, suitable for
creating
   -- a Tcl command, which will increment
the
   -- timer.
   function Tick_Command (
      ClientData : in Integer;
      Interp     : in Tcl.Interp_Ptr;
      Argc    : in C.Int;
      Argv    : in C_Aux.Chars_Ptr_Ptr)
      return C.Int is
   begin -- Tick_Command
      Tick;
      return Tcl.OK;
   end Tick_Command;
   pragma Convention (C, Tick_Command);
```

Figure 13

This alternative illustrates how many system capabilities may be solved with either Tcl- or Ada-centric methods while still taking advantage of the platform independence of the Tk graphical user interface system.

The **Start** command procedure is shown in Figure 14. It starts the timer (if it is not already started) by calling the **Tick** procedure (Figure 12). This function is made into a Tcl command procedure so that it may serve as the Tcl command invoked when the **Start** button is pressed.

```
   -- Declare a procedure, suitable for
creating
   -- a Tcl command, which will start the
timer
   -- if it is currently stopped.  Also,
change
   -- the Stop button (currently labeled
   -- "Reset") to display "Stop."
   function Start_Command (
      ClientData : in Integer;
      Interp     : in Tcl.Interp_Ptr;
      Argc    : in C.Int;
      Argv    : in C_Aux.Chars_Ptr_Ptr)
      return C.Int is
   begin -- Start_Command
      if Stopped then
         Stopped := False;
         Tk.Configure (Stop_Button,
           "-text Stop -command Stop");
         Tick;
      end if;
      return Tcl.OK;
   end Start_Command;
   pragma Convention (C, Start_Command);
```

Figure 14

Figure 15 shows the **Stop** command procedure. It stops incrementing the timer and converts the **Stop** button to a **Reset** button. This function is made into a Tcl command procedure by adhering to the correct subprogram specification and return code conventions. It is used as the command executed when the **Stop** button is pressed.

```
   -- Declare a procedure, suitable for
creating
   -- a Tcl command, which will stop
   -- incrementing the timer.  Also, relabel
   -- the Stop button to be a Reset button.
   function Stop_Command (
      ClientData : in Integer;
      Interp     : in Tcl.Interp_Ptr;
      Argc    : in C.Int;
      Argv    : in C_Aux.Chars_Ptr_Ptr)
      return C.Int is
   begin -- Stop_Command
      Stopped := True;
      Tk.Configure (Stop_Button,
         "-text Reset -command Reset");
      return Tcl.OK;
   end Stop_Command;
   pragma Convention (C, Stop_Command);
```

Figure 15

Resetting the timer value to zero is taken care of by the **Reset** procedure shown in Figure 16. This is another utility procedure that can be called from the **Reset** Tcl command procedure as well as from other Ada procedures. Restting the timer with this procedure also updates the screen to reflect the new timer value.

```
   -- Reset the timer's value to 0.0 and
update
   -- the display.
   procedure Reset is
   begin -- Reset
      Seconds    := 0;
      Hundredths := 0;
      Stopped    := True;
      Update;
   end Reset;
```

Figure 16

Figure 17 shows the **Reset** procedure encapsulated in the interface necessary to make it into a Tcl callable command procedure. **Reset_Command** will be called whenever the **Reset** button is pressed. It uses the utility procedure, **Reset**, to reset the timer to value 0.0 seconds.

```
   -- Declare a procedure, suitable for
creating
   -- a Tcl command, which will reset the
timer
   -- to 0.0 and update the display.
   function Reset_Command (
      ClientData : in Integer;
      Interp     : in Tcl.Interp_Ptr;
      Argc    : in C.Int;
      Argv    : in C_Aux.Chars_Ptr_Ptr)
       return C.Int is
   begin -- Reset_Command
      Reset;
      return Tcl.OK;
   end Reset_Command;
   pragma Convention (C, Reset_Command);
```

Figure 17

Several important operations are shown in the code in Figure 18. The first three lines of code create a Tcl interpreter and initialze Tcl and Tk. These correspond to the "Create Interpreter" and "Initialize Tcl and Tk" boxes of Figure 9. There is no equivalent code in the Tcl/Tk version of the timer program although they are a critical part of the Tcl and Tk C public library interfaces. These lines would be required in a C version of the timer as well in this Ada version. The C interface to Tcl/Tk requires these calls in order to provide the flexibility (not available in Tcl script code) of creating and using several Tcl interpreters. A Tcl interpreter preserves the execution state of a Tcl script, including commands implemented in C or Ada.

```
begin -- Timer

   -- Create one Tcl interpreter and
initialize
   -- Tcl and Tk.
   Interp := Tcl.CreateInterp;
   Result := Tcl.Init (Interp);
   Result := Tk.Init  (Interp);

   -- Create several new Tcl commands to
call
   -- Ada subprograms.
   CreateCommands.CreateCommand (Interp,
      "tick",  Tick_Command'access,  0,
NULL);
   CreateCommands.CreateCommand (Interp,
      "Start", Start_Command'access, 0,
NULL);
   CreateCommands.CreateCommand (Interp,
      "Stop",  Stop_Command'access,  0,
NULL);
   CreateCommands.CreateCommand (Interp,
      "Reset", Reset_Command'access, 0,
NULL);

   -- Set both Tcl and Tk contexts so that
   -- we may use shortcut Tk calls that
require
   -- reference to the interpreter.
   Tcl.Set_Context (Interp);
   Tk.Set_Context  (Interp);
```

Figure 18

The next section of Figure 18 consists of four calls to **CreateCommand** to register Ada subprograms as Tcl command procedures. This must be done in the context of a specific Tcl interpreter and each procedure must be given a unique command name. Thereafter, whenever this name is referenced in the specified interpreter as a command, the corresponding Ada procedure is called by the Tcl interpreter. The interpreter parses the arguments of the command and passes them to the Ada subprogram via the **Argv** argument shown in each of the subprogram declarations (Figures 13, 14, 15, and 17).

The last two lines of code have no equivalent in either Tcl script code or in the Tcl Library. They were added as part of the Ada "thick" binding to Tcl and Tk. The purpose of these is to provide a shortcut to calling Tk procedures which require an interpreter argument. The Widget tagged type, upon which all Ada Tk widget types are based, contains a reference to the interpreter so that calls which include a widget argument need not also specify an interpreter. Other calls in Tk require an interpreter argument but do not require a widget argument. Setting the context interpreter for Tcl and Tk allows the programmer a shortcut for specifying the interpreter. Appendix B shows that there are several subprograms available in the Tk Ada interface which allow the programmer to set and get the context interpreter.

Figure 19 presents the code which creates the three subwindows to display the timer value and to start, stop, and reset the timer. For each widget, we specify its name and several attributes which control its appearance and location in the main Tk window. For the two button widgets, we also specify the Tcl command procedure to be called when the button is invoked. Recall that these procedures are actually Ada functions which were registered in Figure 18 to act as Tcl command procedures.

```
   -- Create and pack the counter text
widget
   Counter := Tk.Create (".counter",
      "-text 0.00 -relief raised -width
10");
   Tk.Pack (Counter, "-side bottom -fill
both");

   -- Create and pack the Start button
   Start_Button := Tk.Create (".start",
      "-text Start -command Start");
   Tk.Pack (Start_Button,
      "-side left -fill both -expand yes");

   -- Create and pack the Stop button
   Stop_Button := Tk.Create (".stop",
      "-text Reset -command Reset");
   Tk.Pack (Stop_Button,
      "-side left -fill both -expand yes");
```

Figure 19

Figure 20 completes the Ada version of the timer program. Two statements bind keys to event sequences. Both the Control-C and Control-Q keys will destroy the main Tk window, then exit the program. It is not shown here, but if any clean-up were needed, we could have coded another Ada subprogram as a Tcl command procedure and called it from within the Bind script action argument. Appendix B shows that the Tk package also supports calls to bind to any widget as well as to destroy a binding.

```
   -- Bind ^C and ^Q keys to exit
   Tk.Bind_to_Main_Window (Interp,
      "<Control-c>", "{destroy .;exit}");
   Tk.Bind_to_Main_Window (Interp,
      "<Control-q>", "{destroy .;exit}");

   -- Loop inside Tk, waiting for commands
to
   -- execute.  When there are no windows
left,
   -- Tk.MainLoop returns and we exit.

   Tk.MainLoop;

end Timer;
```

Figure 20

Notice that the **`Reset`** procedure is not called in the Ada version of the timer program as it is in the Tcl version (Figure 8). Since the variables were already initialized in their declarations and the timer value of 0.0 was already displayed (in the creation of **`.counter`** label widget in Figure 19), this code is not necessary in the Ada version. This takes advantage of improved reliability offered by Ada in variable initialization during elaboration. References to uninitialized variables is a very common problem in development of Tcl script code.

Finally, the Ada version explicitly turns over control to the Tk event loop handler. This is done implicitly in the Tcl version when the Tcl script interpreter encounters the end of the script file.

Notice how easy it is to mix Ada code with Tcl code. For example, the timer value is maintained in Ada variables, **`Seconds`** and **`Hundredths`**, not Tcl variables. Figure 12 shows the **`Tick`** procedure which updates the time with Ada rather than Tcl statements.

The timer program example is not CPU-intensive or a very complex application. In a "real" application, preference for use of Ada over Tcl where possible can yield much of the benefits of program reliability and efficiency for which Ada was designed [3] and for which Tcl is not well known. With the addition of Tcl/Tk through the TASH binding, application development can benefit from the use of a platform-independent graphical user interface toolkit as well as many other Tcl features.

Major Design Decisions

Several major design decisions were made while developing these bindings. These decisions are summarized here and described in more detail in the following sections.

- Whether to translate the Tk C Library interface directly or provide a higher level binding to Tk,
- Whether to use **`Tcl.Eval`** to execute Tk commands or to create interfaces to the Tk commands,
- Whether to use object-oriented techniques,
- Whether to use child library units or one single package,
- Whether to provide Tk widget attributes as strings or enumeration types.

A "Thick" Binding

The Tcl binding described in [8] is an enhanced translation of the Tcl C Library as represented in the header file, **`tcl.h`**. The enhancements increase the "thickness" of the binding by adding subprograms which use standard Ada data types and exception handling to the primitive C functions.

In addition, translation of every Tcl command procedure to an Ada-callable function was considered for inclusion in the binding.

This was not done in the first release because the public interface to the Tcl system was considered adequate to satisfy the immediate goals of the Ada Tcl binding, that is, to allow a Tcl program to use Ada in place of C for defining new Tcl command procedures and to make the Tcl library functions available to an Ada program.

Some Tcl command procedures will never need to be implemented in the Ada binding. For example, conditional and loop statements are not needed since they are done much more effectively in Ada. The remaining commands that do not have direct representation in the Tcl library functions can, for now, be easily implemented with calls to **`Tcl.Eval`**. List processing commands, such as **`lindex`** and **`linsert`**, are examples. This was done in the intial release to speed development. Future work is planned in this area to provide a complete interface without having to resort to **`Tcl.Eval`**.

The opposite approach has been taken with the Tk binding. The primary purpose of the Tk C Library is to allow creation of new widgets. Although important, this was not considered as valuable for the first implemented version of the Ada interface as the ability to utilize the predefined widgets, pack them into windows, and provide for user interaction. Very few of the Tk command procedures to perform these functions are represented in the Tk C Library interface. So, the Ada binding implements the Tk command procedures rather than facilities of the **`tk.h`** header file.

Several of the **`tk.h`** functions, such as **`Tk_Init`**, **`Tk_Main`**, **`Tk_MainLoop`**, are essential to getting a Tk program to operate and so are included in the binding.

Why not use Tcl.Eval?

But, why not allow the Ada programmer to create Tk windows by simply calling **`Tcl.Eval`**? As long as the Tk library is linked with the Ada program and the Tk initialization procedures are called, this works quite effectively.

One reason for providing the procedures is that, where useful, multiple Tcl interpreters can be used. Each window must be created in the context of a Tcl interpreter. Multiple interpreters can be created with the Tcl binding, although one will be adequate for most situations.

Another reason is that window path names are often stored in variables. This can make the code cleaner since path names often get very long. Even though this implementation requires path name variables to remain strings, it is more efficient to store them in Ada variables than in Tcl variables. Use of Tcl variables would require a hash table lookup to find the variable each time it is referenced in a **`Tcl.Eval`** call.

So, primarily to enhance efficiency, it was decided to provide an Ada interface to each Tk command procedure rather than rely exclusively on **`Tcl.Eval`**.

Use of Object-Oriented Techniques

The X Window System has been promoted as an object-oriented system in spite of the fact that it is implemented in C, a language which is neither object-oriented or even object-based. Heller [2] writes that

> XView is an *object-oriented* toolkit. XView objects can be considered building blocks from which the user interface of the application is assembled. Each piece can be considered an *object* from a particular *package*.

Lately, user interface toolkits, such as Fresco [9], have been implemented which provide a true object-oriented interface to the X Window System.

Tk, however, is not an object-oriented toolkit. The first version of this Ada binding was not object-oriented. This was done primarily because Tcl/Tk itself is not object-oriented. In spite of this, the current version of the Ada binding to Tk uses tagged types to implement the screen widget types. It also takes advantage of inheritance in several places to develop the widget class hierarchy. Figure 21 shows the widget hierarchy as implemented in the Tk interface in Appendix B.

The decision was made to use object-oriented techniques for the Ada binding to Tk primarily because this is intended to be a “thick” binding to make a user interface toolkit available to Ada programmers and most newly developed toolkits with which this toolkit might “compete” are using object-oriented techniques.

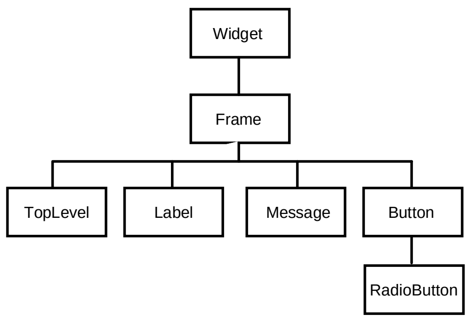

Figure 21

Use of Child Library Units

Early versions of this Tk binding used a separate child library package for each Tk command. Originally, it was thought that this would reduce complexity by allowing the programmer to focus on only those commands necessary for a particular job.

However, this was changed for several reasons:

- A programmer will typically learn Tk from a reference such as [6] rather than from reading the Ada Tk binding. Therefore, separating the commands will not necessarily reduce the complexity of choosing the right widget for each situation. On the contrary, it might increase it.
- Since this is a binding to a C library, separate child packages will not, by itself, reduce the final object size.
- Real programs typically use many different types of widgets. The need to “with” a package for each will turn out to be a burden rather than helpful.

Therefore, the current version of this binding is encapsulated in one single package.

Use of Enumeration Types for Widget Attributes

In many graphical user interfaces, attributes are used to control the appearance and behaviour of the widgets. This seems like a natural application of Ada enumeration types. Use of enumeration types may reduce the chance of error in setting attributes by limiting what attributes may be used in each tk command. This would allow compile-time checking to catch attribute misuse.

However, this usage was decided against because:

- It was judged that veteran Tk programmers would dislike converting existing and new code to this method and first-time Tk programmers are likely to learn the attributes from Tk reference material, not this binding. In either case, the gain was not considered adequate to radically change normal Tk attribute usage.

- Although arbitrary lists of arguments (such as attributes) can be constructed in Ada, their usage is generally very cumbersome. The added complexity of creating such lists was considered to outweigh the benefits of the use of enumeration types.
- Since this is a binding, it must be updated each time a new version of Tk is introduced. The maintenance task of keeping up with the addition of new widget attributes was considered to be potentially very burdensome in comparison to the benefits of the use of enumeration types.

Therefore, this binding retains the familiar Tk style and provides for a string argument of arbitrary length to be used to set attributes.

Future plans

Future development on TASH will include upgrading it to Tcl 7.5 and Tk 4.1 and completing the addition of remaining Tcl and Tk command procedures so that the are directly callable from Ada.

Trademarks and Copyrights

The Tcl/Tk source code is copyrighted by the Regents of the University of California and Sun Microsystems, Inc.

The X Window System is a trademark of the Massachusetts Institute of Technology.

XView is a trademark of Sun Microsystems, Inc.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

Windows and Windows 95 are trademarks of Microsoft, Inc.

References[4]

1. Harnack, Andrew, and Gene Kleppinger. "Beyond the MLA Handbook: Documenting Electronic Sources on the Internet." 10 June, 1996. <http://falcon.eku.edu/honors/beyond-mla> (4 Aug 1996).
2. Heller, Dan. *XView Programming Manual*. O'Reily & Associates, Sebastopol, CA, 1990.
3. Intermetrics. *Ada Reference Manual*. Intermetrics, Cambridge, MA, 1995.
4. McKay, Ross. "The Platform-Independent Graphical User Interface Frequently Asked Questions". 29 May 1996. <http://www.zeta.org.au/~rosko/pigui.htm> (5 Aug 1996).
5. New York University. "The GNAT Project." 1 July 1995. <http://cs.nyu.edu/cs/projects/gnat> (5 Aug 1996).
6. Ousterhout, John. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, MA, 1994.
7. Ousterhout, John. "The Tcl/Tk Project at Sun Microsystems." 16 May 1996. <http://www.smli.com/research/tcl> (5 Aug 1996).
8. Westley, Terry. "TASH: Tcl Ada SHell, An Ada/Tcl Binding." *ACM SIGAda Ada Letters*, 1996.
9. X Consortium. "Fresco -- A Fresh Approach to User Interface Systems." 25 July 1996. <http://www.faslab.com/fresco/HomePage.html (5 Aug 1996).

[4]Referenced documents which reside on the Internet are included in the references in the style recommended in [1].

Appendix A -- How to get and install TASH

Get TASH via anonymous ftp from URL **ftp://ocsystems.com/xada/tash/tash<version-number>.tar.gz**, where <version-number> denotes a version number, e.g. 2.1b1.

Uncompress and extract it from the tar archive:

```
  gzcat tash2.1b1.tar.gz | tar xvf -
```

Then, follow these steps to build and test it:

1. Modify the file, **Makefile.common** in the **tash2.1b1** directory to reference the correct location of your local Tcl and Tk C libraries, **libtcl.a** and **libtk.a**.
2. Type **make** in the **tash2.1b1** directory. This executes a make in each of three subdirectories to build the Ada/Tcl interface, the Tcl Ada Shell program, **tash**, the **freq** demo program, and a test program.
3. Test that the system was correctly built by changing to the **tash2.1b1/test** directory and executing **make test**.
4. Try the **freq** demo by executing **make test** in the **tash2.1b1/demos** directory. To compare the execution time of Tcl versus Ada **freq**, execute **make time**.

Appendix B -- The Tk Package

This is a sample specification of the Tk package. This specification is preliminary and incomplete. It does not include many of the Tk widgets. It serves primarily to show the intent and design of the completed interface to Tk.

```
--------------------------------------------
----
--
-- tk.ads --
--
--     This is an Ada 95 interface to Tk
--     version 4.1a2.
--
-- Copyright (c) 1987-1994
-- The Regents of the University of
California.
--
-- Copyright (c) 1994-1995
-- Sun Microsystems, Inc.
--
-- Copyright (c) 1995-1996
-- Terry J. Westley
--
-- See the file "license.terms" for
information
-- on usage and redistribution of this file,
-- and for a DISCLAIMER OF ALL WARRANTIES.
--
--------------------------------------------
----

with C_Aux;
with Interfaces.C.Strings;
with Tcl;

package Tk is

   Tk_Version  : constant String := "4.1a2";
   Ada_Version : constant String := "2.1b1";

   package C renames Interfaces.C;

   -----------------------------------------
----
   --
   --  The Widget data type, parent of all
   --  objects displayed on the screen.
   --
   --  It is abstract because it is just a
   --  convenience for creating a Widget
   --  class and for creating non-abstract
   --  derived widget types.  Since there
   --  is no such data type in Tk, we make
   --  it abstract so that no instance of
   --  type Widget may be created.
   --
   -----------------------------------------
----

   type Widget is abstract tagged private;

   -----------------------------------------
----
   --
   --  Widget path name constructors
   --
   -----------------------------------------
----

   function Widget_Image (
      Win : in Widget'Class) return String;
   -- Returns the string name of Win.

   function "&" (
      Left  : in Widget'Class;
      Right : in Widget'Class) return
String;
   function "&" (
      Left  : in Widget'Class;
      Right : in String) return String;
   function "&" (
      Left  : in String;
      Right : in Widget'Class) return
String;
   -- Concatenates and returns the string
   -- names of Left and Right.  Does not
   -- insert the separating dot.

   pragma Inline (Widget_Image, "&");

   procedure Set_Context (
      Interp : in Tcl.Interp_Ptr);
   -- Sets the interpreter context for all
Tk
   -- calls which do not include either an
   -- Interp or Widget parameter.

   function Get_Context return
Tcl.Interp_Ptr;
   -- Gets the current interpreter context.

   function Get_Interp (
      Widgt : in Widget'Class)
      return Tcl.Interp_Ptr;
   -- Gets the interpreter of the specified
   -- Widget.

   -----------------------------------------
----
   --
   --  Widget constructors
   --
   -----------------------------------------
----

   function Create (
      pathName : in String;
      options  : in String := "")
      return Widget is abstract;
   -- Creates a new widget in the
"contextual"
   -- interpreter.  Options may be specified
   -- via the "options" parameter or the
option
   -- database to configure the widget.

   function Create (
      Interp   : in Tcl.Interp_Ptr;
      pathName : in String;
      options  : in String := "")
      return Widget is abstract;
   -- Creates a new widget in the specified
   -- interpreter.  Options may be specified
   -- via the "options" parameter or the
option
   -- database to configure the widget.

   -----------------------------------------
----
   --
   --  Widget destructor
   --
   -----------------------------------------
----

   procedure Destroy (
      Widgt : in out Widget'Class);
   -- Destroys a widget.

   -----------------------------------------
----
   --
   --  Widget configuration query and modify
   --
   -----------------------------------------
----

   function cget (
      Widgt  : in Widget'Class;
      option : in String) return String;
   -- Returns the current value of the
   -- specified configuration option.

   function configure (
      Widgt   : in Widget'Class;
      options : in String := "") return
String;
   procedure configure (
      Widgt   : in Widget'Class;
      options : in String := "");
   -- Queries or modifies the configuration
   -- options.  If options is an empty
string,
   -- returns a list of all available
options
   -- for the widget.

   -----------------------------------------
----
   --
   --  Bind associates a Tcl script with an
   --  event.  The script is executed when
   --  the event occurs.
   --
   -----------------------------------------
----

   procedure Bind (
      Widgt    : in Widget'class;
      Sequence : in String;
      Script   : in String);
   -- Associates Tcl script Script with the
   -- event Sequence.

   procedure Bind (
      Widgt    : in Widget'class;
      Sequence : in String);
   function Bind (
      Widgt    : in Widget'class;
      Sequence : in String) return String;
   -- Disassociates the binding from the
event
   -- Sequence.

   procedure Bind_to_Main_Window (
      Interp   : in Tcl.Interp_Ptr;
      Sequence : in String;
      Script   : in String);
   -- Associates Tcl script Script with the
   -- event Sequence in the main window.

   procedure Bind_to_Main_Window (
      Interp   : in Tcl.Interp_Ptr;
      Sequence : in String);
   function Bind_to_Main_Window (
      Interp   : in Tcl.Interp_Ptr;
      Sequence : in String) return String;
   -- Disassociates the binding from the
event
   -- Sequence in the main window.

   -----------------------------------------
----
   --
   --  Frame widget
   --
   --  This is a non-abstract type derived
   --  directly from Widget.  Each of the
   --  derived widgets redefines the Create
   --  subprogram in order to create the
   --  correct type of widget.
   --
   -----------------------------------------
----

   type Frame is new Widget with private;

   function Create (
      pathName : in String;
      options  : in String := "") return
Frame;
   -- Creates a new widget in the
"contextual"
   -- interpreter and makes it into a frame
   -- widget.  Options may be specified via
the
   -- "options" parameter or the option
database
   -- to configure the widget.

   function Create (
      Interp   : in Tcl.Interp_Ptr;
      pathName : in String;
      options  : in String := "") return
Frame;
   -- Creates a new widget in the specified
   -- interpreter and makes it into a frame
   -- widget.  Options may be specified via
the
   -- "options" parameter or the option
database
   -- to configure the widget.

   -----------------------------------------
----
   --
   --  Toplevel widget
   --
   -----------------------------------------
----

   type Toplevel is new Frame with private;

   function Create (
      pathName : in String;
      options  : in String := "")
      return Toplevel;
   -- Creates a new widget in the
"contextual"
   -- interpreter and makes it into a
toplevel
   -- widget.  Options may be specified via
the
   -- "options" parameter or the option
database
   -- to configure the widget.

   function Create (
      Interp   : in Tcl.Interp_Ptr;
      pathName : in String;
      options  : in String := "")
      return Toplevel;
   -- Creates a new widget in the specified
   -- interpreter and makes it into a
toplevel
   -- widget.  Options may be specified via
the
   -- "options" parameter or the option
database
   -- to configure the widget.

   -----------------------------------------
----
   --
   --  Label widget
   --
   -----------------------------------------
----

   type Label is new Frame with private;

   function Create (
      pathName : in String;
      options  : in String := "") return
Label;
   -- Creates a new widget in the
"contextual"
   -- interpreter and makes it into a label
   -- widget.  Options may be specified via
the
   -- "options" parameter or the option
database
   -- to configure the widget.

   function Create (
      Interp   : in Tcl.Interp_Ptr;
      pathName : in String;
      options  : in String := "") return
```

```
Label;
   -- Creates a new widget in the specified
   -- interpreter and makes it into a label
   -- widget.  Options may be specified via
the
   -- "options" parameter or the option
database
   -- to configure the widget.

   -----------------------------------------
----
   --
   --  Message widget
   --
   -----------------------------------------
----

   type Message is new Frame with private;

   function Create (
      pathName : in String;
      options  : in String := "")
      return Message;
   -- Creates a new widget in the
"contextual"
   -- interpreter and makes it into a
message
   -- widget.  Options may be specified via
the
   -- "options" parameter or the option
database
   -- to configure the widget.

   function Create (
      Interp   : in Tcl.Interp_Ptr;
      pathName : in String;
      options  : in String := "")
      return Message;
   -- Creates a new widget in the specified
   -- interpreter and makes it into a
message
   -- widget.  Options may be specified via
the
   -- "options" parameter or the option
database
   -- to configure the widget.

   -----------------------------------------
----
   --
   --  Button widget
   --
   -----------------------------------------
----

   type Button is new Frame with private;

   function Create (
      pathName : in String;
      options  : in String := "") return
Button;
   -- Creates a new widget in the
"contextual"
   -- interpreter and makes it into a button
   -- widget.  Options may be specified via
the
   -- "options" parameter or the option
database
   -- to configure the widget.

   function Create (
      Interp   : in Tcl.Interp_Ptr;
      pathName : in String;
      options  : in String := "") return
Button;
   -- Creates a new widget in the specified
   -- interpreter and makes it into a button
   -- widget.  Options may be specified via
the
   -- "options" parameter or the option
database
   -- to configure the widget.

   procedure Flash (
      Buttn : in Button'class);
   -- Flash the button.

   function Invoke (
      Buttn : in Button'class;
      options  : in String := "") return
String;
   -- Invoke the Tcl command associated with
   -- the button.

   -----------------------------------------
----
   --
   --  RadioButton widget
   --
   -----------------------------------------
----

   type RadioButton is new Button with
private;

   function Create (
      pathName : in String;
      options  : in String := "")
      return Radiobutton;
   -- Creates a new widget in the
"contextual"
   -- interpreter and makes it into a
   -- radiobutton widget.  Options may be
   -- specified via the "options" parameter
or
   -- the option database to configure the
   -- widget.

   function Create (
      Interp   : in Tcl.Interp_Ptr;
      pathName : in String;
      options  : in String := "")
      return Radiobutton;
   -- Creates a new widget in the specified
   -- interpreter and makes it into a
   -- radiobutton widget.  Options may be
   -- specified via the "options" parameter
or
   -- the option database to configure the
   -- widget.

   procedure Deselect (
      Buttn : in RadioButton);
   -- Deselect the button.

   procedure Tk_Select (
      Buttn : in RadioButton);
   -- Select the button.

   procedure Toggle (
      Buttn : in RadioButton);
   -- Toggle the button.

   -----------------------------------------
----
   --
   --  After commands
   --
   --  These commands delay execution and
   --  schedule (and unschedule) future
   --  execution of Tcl commands.
   --
   -----------------------------------------
----

   procedure After (
      Ms     : in Natural);
   -- Sleeps for Ms milliseconds in the
   -- "contextual" interpreter.

   procedure After (
      Interp : in Tcl.Interp_Ptr;
      Ms     : in Natural);
   -- Sleeps for Ms milliseconds in the
   -- specified interpreter.

   function  After (
      Ms     : in Natural;
      Script : in String) return String;
   procedure After (
      Ms     : in Natural;
      Script : in String);
   -- Arranges for the Tcl Script to be
   -- executed after Ms milliseconds in the
   -- "contextual" interpreter.  The
function
   -- returns an identifier suitable for
   -- canceling the command.

   function  After (
      Interp : in Tcl.Interp_Ptr;
      Ms     : in Natural;
      Script : in String) return String;
   procedure After (
      Interp : in Tcl.Interp_Ptr;
      Ms     : in Natural;
      Script : in String);
   -- Arranges for the Tcl Script to be
   -- executed after Ms milliseconds in the
   -- specified interpreter.  The function
   -- returns an identifier suitable for
   -- canceling the command.

   procedure Cancel (
      id_or_script : in String);
   -- Cancels the execution of a delayed
   -- command in the "contextual"
interpreter.

   procedure Cancel (
      Interp       : in Tcl.Interp_Ptr;
      id_or_script : in String);
   -- Cancels the execution of a delayed
   -- command in the specified interpreter.

   function  Idle (
      Script : in String) return String;
   procedure Idle (
      Script : in String);
   -- Arranges for the Tcl Script to be
   -- executed later as an idle handler in
the
   -- "contextual" interpreter.  The
function
   -- returns an identifier suitable
   -- for canceling the command.

   function  Idle (
      Interp : in Tcl.Interp_Ptr;
      Script : in String) return String;
   procedure Idle (
      Interp : in Tcl.Interp_Ptr;
      Script : in String);
   -- Arranges for the Tcl Script to be
   -- executed later as an idle handler in
the
   -- specified interpreter.  The function
   -- returns an identifier suitable for
   -- canceling the command.

   function Info (
      id     : in String := "") return
String;
   -- Returns information about existing
event
   -- handlers in the "contextual"
interpreter.

   function Info (
      Interp : in Tcl.Interp_Ptr;
      id     : in String := "") return
String;
   -- Returns information about existing
event
   -- handlers in the "contextual"
interpreter.

   -----------------------------------------
----
   --
   --  Pack commands
   --
   --  These commands provide for packing
   --  widgets within other widgets and
   --  therefore rendering them to the
screen.
   --
   -----------------------------------------
----

   procedure Pack (
      Slave   : in Widget'Class;
      Options : in String);
   procedure Pack_Configure (
      Slave   : in Widget'Class;
      Options : in String);
   -- Tells the packer how to configure the
   -- specified Slave window.

   procedure Pack_Forget (
      Slave   : in Widget'Class);
   -- Removes the Slave window from the
   -- packing list for its master and unmaps
   -- their windows.

   function Pack_Info (
      Slave   : in Widget'Class) return
String;
   -- Returns a list whose elements are the
   -- current configuration state of the
   -- specified Slave window.

   procedure Pack_Propogate (
      Master  : in Widget'Class;
      State   : in Boolean);
   -- Enables or disables propogation for
the
   -- specified Master window.

   function Pack_Propogate (
      Master  : in Widget'Class) return
Boolean;
   -- Returns state of propogation in the
   -- specified Master window.

   function Pack_Slaves (
      Master  : in Widget'Class) return
String;
   -- Returns a list of slaves in the
packing
   -- order of the specified Master window.

   ----------------------------------------
----
   --
   --  tk.h functions
   --
   --  This is a "thin" binding to tk.h
   --  functions.
   --
   ----------------------------------------
----

   function Init (
      interp          : in Tcl.Interp_Ptr)
      return C.Int;
   pragma Import (C, Init,
              "Tk_Init");

   procedure Main (
      argc            : in C.Int;
      argv            : in
C_Aux.Chars_Ptr_Ptr;
      appInitProc     : in
Tcl.AppInitProc_Ptr);
   pragma Import (C, Main,
              "Tk_Main");

   procedure DoOneEvent (
      flags           : in C.Int);
   pragma Import (C, DoOneEvent,
              "Tk_DoOneEvent");

   procedure MainLoop;
   pragma Import (C, MainLoop,
"Tk_MainLoop");

   function GetNumMainWidgets return C.Int;
   pragma Import (C, GetNumMainWidgets,
              "Tk_GetNumMainWidgets");

private

   type Widget is abstract tagged
      record
         Name   : C.Strings.Chars_Ptr;
        Interp : Tcl.Interp_Ptr;
      end record;

   Context : Tcl.Interp_Ptr;

   procedure Execute_Widget_Command (
      Widgt   : in Widget'Class;
      command : in String;
      options : in String := "");

   type Frame is new Widget
      with null record;

   type Toplevel is new Frame
      with null record;

   type Label is new Frame
      with null record;

   type Message is new Frame
      with null record;

   type Button is new Frame
      with null record;

   type RadioButton is new Button
      with null record;

end Tk;
```