The New [incr Tcl]: Objects, Mega-Widgets, Namespaces and More

Michael J. McLennan
AT&T Bell Laboratories

The New `[incr Tcl]`: Objects, Mega-Widgets, Namespaces and More

Michael J. McLennan

AT&T Bell Laboratories
1247 S. Cedar Crest Blvd.
Allentown, PA 18103
michael.mclennan@att.com

Abstract

The allure of using Tcl/Tk is the way that applications come together with relative ease. A sticky note facility can be put together in an hour. A simple video game can be created in an afternoon. But as applications get larger, Tcl/Tk code becomes more and more difficult to understand, maintain and extend. [incr Tcl] provides the extra language support needed to build large Tcl/Tk applications. The latest release offers better performance and a host of new features. The global namespace of commands and variables can now be partitioned into smaller namespaces acting as subsystems. The class facility extends the concept of namespaces to support unique instances of objects with similar characteristics. This technology can be used to create mega-widgets and other high-level building blocks that help applications come together even faster than before, with better structure in the resulting code.

Introduction

Object-oriented programming is not an alternative to procedural programming, it is an evolution beyond it. Indeed, object-oriented programming is full of procedures, but these procedures are organized around the data in the application, in a way that makes the system easier to maintain and extend.

`[incr Tcl]` is an object-oriented extension of the Tcl language. It was conceived two years ago [1] to provide better support for building large Tcl/Tk applications. Since then, it has been used to build support software for telephone communication and cellular phone networks. It is the backbone of a distributed computational chemistry system. It controls a particle accelerator. It supports software on Wall Street. It even supports the flight software that will take the Pathfinder to Mars [2].

The latest release of `[incr Tcl]` extends the Tcl language even further. It has been redesigned and rewritten to provide better packaging facilities and a host of new features:

◆ ***NAMESPACES***

`[incr Tcl]` contains a namespace facility patterned after the proposal by Howlett [3]. It can be used to package data, procedures, classes, and other namespaces into reusable building blocks that are easily integrated into other applications.

◆ ***MEGA-WIDGETS***

`[incr Tk]` provides a framework for combining Tk widgets into a larger building block. The resulting widget looks and acts like a regular Tk widget, but provides a higher level of functionality.

◆ ***PUBLIC / PROTECTED / PRIVATE***

The accessibility of both procedures and data can be "public" (open to other classes/namespaces), "protected" (open only to friendly classes/namespaces), or "private" (closed to all other classes/namespaces).

◆ ***SUPPORT FOR INTEGRATING C/C++ CODE***

Procedures can be implemented with Tcl code, or with C/C++ code.

◆ ***DYNAMIC LOADING***

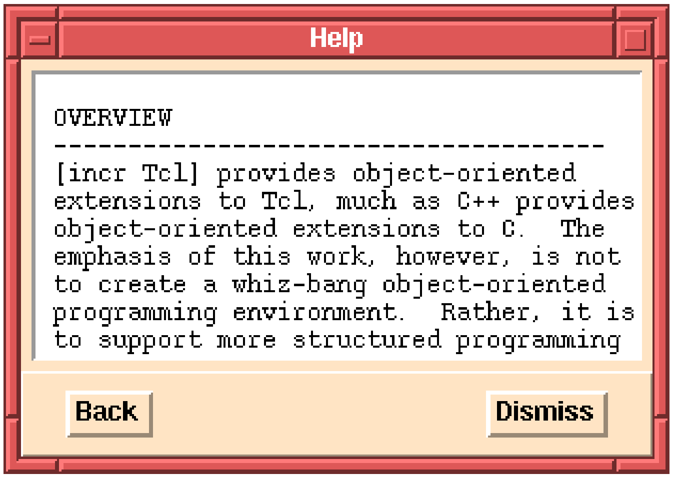
C/C++ functions and extensions can be dynamically loaded on architectures that support shared objects and libraries.

◆ ***MORE DYNAMIC CLASSES***

The bodies of class methods and procs can be redefined at any point, making it easier to do interactive debugging and development.

◆ ***BETTER PERFORMANCE***

The performance of `[incr Tcl]` is now on par with Tcl, and memory consumption has been drastically reduced.



```
OVERVIEW
-------------------------------------
[incr Tcl] provides object-oriented
extensions to Tcl, much as C++ provides
object-oriented extensions to C.  The
emphasis of this work, however, is not
to create a whiz-bang object-oriented
programming environment.  Rather, it is
to support more structured programming
```

Figure 1 – On-line help facility

◆ ***NO MORE MULTIPLE INTERPRETERS***

> Previous versions of `[incr Tcl]` used a separate interpreter to maintain the namespace for each class. The current version layers the class mechanism on top of the namespace facility.

`[incr Tcl]` does not change the fundamental character of Tcl. It does not change the syntax or semantics of the language. Rather, it adds the ability to package the usual variables and procedures into a more understandable and reusable form. In this paper, we show how a simple Tk “help” facility can be transformed into a better building block by packaging it first in a namespace, then as a mega-widget. Finally, we extend its capabilities by introducing supporting classes.

Tk “Help” Facility

Suppose that we have an application that requires some on-line help. It is easy to build such a facility with Tcl/Tk. We could combine a text widget and a few buttons to create the rudimentary help window shown in Figure 1. We will go one step further, and create some procedures to access this facility. The command:

`help_topic` *file*

loads a *file* of help information into the viewer. The help facility should keep track of the list of topics that have been loaded, so that at any point the command

```
help_back
```

will take the user back to the previous help page. We also create the command:

```
help_show
```

that is used internally by `help_topic`, to make sure that the help viewer is visible whenever a new topic is loaded. The code used to implement this facility is shown in Figure 2.

Most of this code is quite straightforward. The procedure `help_show` creates the help window if it does not already exist, and makes it visible on the screen. The procedure `help_topic` reads text from the specified file and loads it into the help viewer. It also adds the file name to the list of topics that have been loaded into the viewer. This list is maintained as a global variable `HelpFiles`, so that it can be accessed by other help procedures. For example, the procedure `help_back` finds the previous file name near the end of the list and loads this into the viewer.

Notice that we have added the prefix “`help_`” to each of these commands. This makes it easier to recognize the relationship between them, and also makes it less likely that these command names will clash with others in the application. Also, we are careful to name global variables like `HelpFiles` with a special “`Help`” prefix. But although this naming convention makes accidental name clashes less likely, it cannot prevent them. Moreover, we cannot guarantee that other procedures in our application will not sabotage our interface and access global variables directly. Suppose that we want to protect certain commands like `help_show` that are used internally by the package, but that are not meant to be part of the public interface. With ordinary Tcl, the best we can hope to do is add a special prefix to the command name (e.g., “`_help_show`”) and discourage its use.

Using Namespaces

`[incr Tcl]` offers a way of packaging together

```
set HelpFiles {}

proc help_show {} {
    if {![winfo exists .help]} {
        toplevel .help
        text .help.info -wrap none -borderwidth 2 -relief sunken
        pack .help.info -side top -fill both -padx 5 -pady 5
        frame .help.cntls -borderwidth 1 -relief raised

        button .help.cntls.back -text "Back" -command help_back
        pack .help.cntls.back -side left -padx 20 -pady 8

        button .help.cntls.dismiss -text "Dismiss" \
            -command {wm withdraw .help}
        pack .help.cntls.dismiss -side right -padx 20 -pady 8
        pack .help.cntls -side bottom -fill x

        wm title .help "Help"
    }
    wm deiconify .help
}

proc help_topic {file} {
    global HelpFiles

    if {[catch "open $file r" fid] != 0} {
        error "can’t open help file \"$file\""
    }
    set info [read $fid]
    close $fid

    help_show
    .help.info configure -state normal
    .help.info delete 1.0 end
    .help.info insert end $info
    .help.info configure -state disabled

    lappend HelpFiles $file
}

proc help_back {} {
    global HelpFiles

    set last [expr [llength $HelpFiles]-2]
    if {$last >= 0} {
        set file [lindex $HelpFiles $last]
        set HelpFiles [lrange $HelpFiles 0 [expr $last-1]]

        help_topic $file
    }
}
```

Figure 2 – Ordinary Tcl/Tk code used to implement the help facility

procedures and global variables, and controlling access to them. Our familiar help facility can be wrapped in the confines of a namespace, as shown in Figure 3. Members that are "public" are accessible from any other namespace, including the global namespace. Members that are "protected" are accessible only if other namespaces request special access. Members that are "private" are completely hidden from other namespaces.

Once a namespace is created, the public members within it can be accessed using the special scope "`::`" qualifier. From the global namespace, a topic can be displayed in our help viewer as follows:

`help::topic` *file*

It is not possible, however, to access the private `HelpFiles` variable from the global namespace, even if the scope qualifier is used:

```
# this will fail:
set help::HelpFiles ""
```

Within the confines of a namespace, procedures and global variables belonging to the namespace can be accessed directly. Within the body of `help::topic`, for example, the command `show` requires no special "`help::`" qualifier.

Any commands that are created within the context of a namespace belong to that namespace. For proc declarations such as those shown in Figure 3, this would be expected. But this also implies that any widget created within a namespace has an access command that belongs to the namespace. For example, the toplevel window `.help` can only be accessed via the "`.help`" command in the `help` namespace:

```
namespace help {
    .help configure -bg gray
}
```

In effect, the access command is protected by the namespace as a detail of its implementation.

Of course, the window name "`.help`" is still recognized everywhere within the application. It is still possible to query information about the widget from any context:

```
set w [winfo reqwidth .help]
set h [winfo reqheight .help]
```

For some applications it may be useful to know where a widget's access command resides. This is referred to as the widget's "locality", and it can be queried as follows:

```
% locality widget .help
::help
```

Commands from one namespace can be integrated into another using the namespace "import" feature. For example, we can integrate the `help` namespace into the global namespace by invoking the following command at the global scope:

```
import add help
```

From this point on, commands like `topic` and `back` can be accessed without the special "`help::`" qualifier, as if they were defined in the global namespace. But if a command like `back` is defined in the global namespace, it will override the command imported from the `help` namespace. The scope qualifier can be used in cases like this, to access a command in a specific namespace.

Note that the `import` command only integrates the public members within a namespace; protected and private members remain hidden. But sometimes namespaces are designed to work in cooperation. It is useful to have members which are available to some namespaces, but hidden from all others. Such members are declared as "protected", and can be accessed by any other namespace that imports in a "protected" mode. For example, if we import the `help` namespace in a "protected" mode at the global namespace:

```
import add {help protected}
```

then we can access the protected command `show` from the global scope. There is no way to import in a "private" mode, so private members always remain hidden from other namespaces.

Suppose that we create another namespace like "`plotter`" which will provide plotting facilities based on the BLT toolkit [4] extensions. Suppose that we want to include on-line help in this facility, but we want to make sure that the "`plotter`" help is separate from any other "`help`" included in the application. We can include the `help` namespace as a child within the `plotter` namespace:

```
namespace plotter {
    namespace help {
        ...same as Figure 3...
    }
}
```

Within the `plotter` namespace, help topics can be viewed using "`help::topic`". But at the global scope this same command must be referenced as "`plotter::help::topic`"; this distinguishes the plotter help facility from another help facility that might be included in the same application.

By default, each namespace imports from its parent in the "public" mode. This is why a command like `set`, which is really defined at the global scope, can be used transparently within a child namespace like `plotter` or `plotter::help`. Parents, on the other hand, do

```
namespace help {
    private variable HelpFiles {}

    protected proc show {} {
        if {![winfo exists .help]} {
            toplevel .help
            text .help.info -wrap none -borderwidth 2 -relief sunken
            pack .help.info -side top -fill both -padx 5 -pady 5
            frame .help.cntls -borderwidth 1 -relief raised

            button .help.cntls.back -text "Back" -command back
            pack .help.cntls.back -side left -padx 20 -pady 8

            button .help.cntls.dismiss -text "Dismiss" \
                -command {wm withdraw .help}
            pack .help.cntls.dismiss -side right -padx 20 -pady 8
            pack .help.cntls -side bottom -fill x

            wm title .help "Help"
        }
        wm deiconify .help
    }

    public proc topic {file} {
        global HelpFiles

        if {[catch "open $file r" fid] != 0} {
            error "can’t open help file \"$file\""
        }
        set info [read $fid]
        close $fid

        show
        .help.info configure -state normal
        .help.info delete 1.0 end
        .help.info insert end $info
        .help.info configure -state disabled

        lappend HelpFiles $file
    }

    public proc back {} {
        global HelpFiles

        set last [expr [llength $HelpFiles]-2]
        if {$last >= 0} {
            set file [lindex $HelpFiles $last]
            set HelpFiles [lrange $HelpFiles 0 [expr $last-1]]

            topic $file
        }
    }
}
```

Figure 3 – Implementing help using the namespace facility in `[incr Tcl]`

not automatically import from their children. If they did, the global scope would revert to a hodgepodge of all (public) members in the application.

Without the import facility, namespaces would be little more than a fancy naming convention, swapping "`help::topic`" for "`help_topic`" in our original example. The import facility, however, allows namespaces to act like little building blocks that can be glued together in different ways to provide more functionality.

Creating Mega-Widgets

Suppose that we want to clone a help window, so that the user can view two help pages side by side. Our `help` namespace was only designed to support one help window. It would be helpful if we could use the namespace facility like a cookie cutter, to create lots of similar but unique objects, each one parameterized by its own bundle of data. This is precisely what the `[incr Tcl]` class facility provides. A class is a special kind of namespace shared by a group of related objects. Each object has its own bundle of data, and is manipulated by special procedures called "methods" defined within the class. One class can inherit the characteristics of another, in much the same way that one namespace can import the commands of another.

But simply defining a class of `HelpWin` objects with methods like `topic` and `back` is not enough. We would like to make these objects look like ordinary Tk widgets, which have configuration options like "`-cursor`" and "`-background`". When we set an option like "`-background`", we would like all of the component widgets within the help window to change color.

`[incr Tk]` provides a framework for building composite "mega-widgets" using `[incr Tcl]` classes. It defines a set of base classes that are specialized to create all other widgets. Among these, `itk::Toplevel` provides all of the functionality needed by a toplevel window. The details of `[incr Tk]` are described elsewhere [5], so we simply provide an example here.

A `HelpWin` class based on `itk::Toplevel` is presented in Figure 4. The "constructor" is invoked whenever a new widget is created. It creates all of the internal components within a specific help window and registers their configuration options on a master list. Each component is created with a symbolic name; for example, "hull" is the symbolic name of the toplevel window representing the widget, which is created automatically by the base class. The window path name for any component can be found by accessing the `itk_component` array defined in the base class. For example:

```
$itk_component(hull)
```

is the window path name for the "hull" component.

Methods are defined within the class in much the same way that commands are defined within a namespace. Variables like the private `helpFiles` list are also declared in a similar manner. But unlike namespaces, the variables declared within a class are created for each object. Each `HelpWin` object, therefore, has its own unique list of help files.

The real power of `[incr Tk]` comes from its automatic management of components and configuration options. Each `HelpWin` object acts like a high-level Tk widget with a master list of configuration options:

```
% HelpWin .help
% .help configure
{-activebackground activeBack-
ground Foreground Black Black} {-
activeforeground activeForeground
Background White White} {-back-
ground background Background White
White} {-borderwidth borderWidth
BorderWidth 0 0} {-cursor cursor
Cursor {} {}} {-font font Font *-
Courier-Medium-R-Normal-*-120-* *-
Courier-Medium-R-Normal-*-120-*}
{-foreground foreground Foreground
Black Black} {-geometry geometry
Geometry {} {}} {-relief relief
Relief flat flat} {-textbg text-
Background Background White White}
{-textfg textForeground Foreground
Black Black} {-width width Width
40 40}
```

When we change the "`-foreground`" and "`-background`" options, the change is automatically propagated to each of the internal components that declared those options with a "keep" statement. All of this functionality comes for free from the base class `itk::Toplevel` via a simple "inherit" statement.

Not Everything is a Mega-Widget

Suppose that we want to extend our help facility to handle more than plain text files. Suppose that we also want to support hypertext markup language (HTML) files, "man" pages, *etc*. We could modify our `topic` method to contain a big "switch" statement, to handle each of the various formats allowed by our system. The problem with such switch statements is that they tend to keep popping up at various places in a program, and they are hard to keep up to date. For example, suppose we add a `search` method, to scan through the text of a

```
class HelpWin {
    inherit itk::Toplevel

    constructor {args} {
        itk_component info {
            text $itk_component(hull).info -wrap none \
                -borderwidth 2 -relief sunken
        } {
            keep -cursor -font -width
            rename -background -textbg textBackground Background
            rename -foreground -textfg textForeground Foreground
        }
        pack $itk_component(hull).info -side top -fill both -padx 5 -pady 5

        itk_component cntls {
            frame $itk_component(hull).cntls -borderwidth 1 -relief raised
        } {
            keep -cursor -background
        }

        itk_component back {
            button $itk_component(hull).cntls.back -text "Back" -command "$this back"
        } {
            keep -cursor -background -foreground \
                -activebackground -activeforeground
        }
        pack $itk_component(hull).cntls.back -side left -padx 20 -pady 8

        itk_component dismiss {
            button $itk_component(hull).cntls.dismiss -text "Dismiss" \
                -command "wm withdraw $this"
        } {
            keep -cursor -background -foreground -activebackground -activeforeground
        }
        pack $itk_component(hull).cntls.dismiss -side right -padx 20 -pady 8

        pack $itk_component(hull).cntls -side bottom -fill x
        wm title $this "Help"

        eval configure $args
    }

    protected method show {} {
        wm deiconify $itk_component(hull)
    }

    public method topic {file} {
        ...similar to Figure 3...
    }

    public method back {} {
        ...similar to Figure 3...
    }

    private variable helpFiles {}
}
```

Figure 4 – Implementing help as an `[incr Tk]` mega-widget

help page and look for keywords. We might need to search each of the page formats in a different manner, hence the need for another big switch statement. Suppose that later on we add another page format to our system. We must be careful to track down each of the switch statements and update it accordingly.

Instead, we can use object-oriented programming to provide the basis for an extensible system of page formats. We define a base class `HelpPage` that acts as a common abstraction for all page formats, as shown in Figure 5. It maintains a variable `text` that contains the raw text for the help page, and provides a method `convert` that will create the commands needed to load information into a text widget. We can create a generic help page as follows:

```
HelpPage page1 "Some help text"
```

and load its contents into a text widget like this:

```
eval [page1 convert .help.info]
```

We create a derived class `HelpFile`, also shown in Figure 5, to represent a plain page of help text loaded from a file. This class inherits all of the text handling capability from `HelpPage`, and simply defines a constructor that will load help information from a specified file. We can create a file page as follows:

```
HelpFile page2 info.txt
```

and load its contents into a text widget as before:

```
eval [page2 convert .help.info]
```

In a similar manner, we could create many other page types, each inheriting its core abstraction from `HelpPage`. More complicated page formats might override the default `convert` method with a specialized version, for example, to parse HTML text and generate the appropriate commands to load that information into a text widget.

We could convert our `HelpWin` mega-widget to work with `HelpPage` objects by updating the `topic` method as shown in Figure 5. This method first checks to see that `page` contains the name of a valid `HelpPage` object, then pops up the help viewer, and loads the internal "info" component (i.e., the help viewer's text widget) with the appropriate information. Each `HelpPage` object knows how to interpret its contents and communicate with the text widget, so there is no need for a switch statement to handle the various page formats.

Notice that the same code (e.g., load help text from a file) would appear in this program regardless of whether we use the "switch" approach or the object-oriented approach. Using a switch statement organizes our code along procedural lines. All of the code associated with a particular page format is scattered throughout the program, buried in the switch statements associated with each operation. In stark contrast, the object-oriented approach collects all of the code needed for each page format into a class definition for that format. The resulting code is easier to understand and maintain.

Not every problem lends itself to an object-oriented solution, but when a problem does, it is foolish to overlook the object-oriented approach.

Conclusion

`[incr Tcl]` provides the language support needed to build large Tcl/Tk applications. The current release offers better performance and a host of new features. Namespaces can be used to package procedures and global variables as reusable building blocks. Classes extend the namespace concept to support unique instances of objects with similar characteristics. `[incr Tk]` provides the infrastructure needed to construct mega-widgets, composite widgets with high-level functionality that look and feel like the usual Tk widgets. Using all of this technology, applications can be put together with high-level building blocks in a fraction of the normal development time, and the resulting code will be easier to understand, maintain and extend.

References

[1] M. J. McLennan, "`[incr Tcl]`: Object-Oriented Programming in Tcl," *Proceedings of the Tcl/Tk Workshop*, University of California at Berkeley, June 10-11, 1993.

[2] D. E. Smyth, "Tcl and Concurrent Object-Oriented Flight Software: Tcl on Mars," *Proceedings of the Tcl/Tk 1994 Workshop*, New Orleans, LA, June 23-25, 1994.

[3] G. A. Howlett, "Packages: Adding Namespaces to Tcl," *Proceedings of the Tcl/Tk 1994 Workshop*, New Orleans, LA, June 23-25, 1994.

[4] G. A. Howlett, *The BLT Toolkit*, available by anonymous ftp from: `ftp.aud.alcatel.com:/tcl`.

[5] M. J. McLennan, "`[incr Tk]`: Building Extensible Widgets with `[incr Tcl]`", *Proceedings of the Tcl/Tk 1994 Workshop*, New Orleans, LA, June 23-25, 1994.

```
class HelpPage {
    constructor {{mesg ""}} {
        setText $mesg
    }

    method convert {widget} {
        set cmd {
            namespace [locality widget WIDGET] {
              WIDGET configure -state normal
              WIDGET delete 1.0 end
              WIDGET insert end {TEXT}
              WIDGET configure -state disabled
            }
        }
        regsub -all WIDGET $cmd $widget cmd
        regsub -all TEXT $cmd $text cmd
        return $cmd
    }

    protected method setText {mesg} {
        set text $mesg
    }

    private variable text {}
}

class HelpFile {
    inherit HelpPage

    constructor {file} {
        if {[catch "open $file r" fid] != 0} {
            error "can’t open help file \"$file\""
        }
        setText [read $fid]
        close $fid
    }
}

class HelpWin {
    ...
    method topic {page} {
        if {[itcl_info objects $page -isa HelpPage] == ""} {
            error "not a help page: $page"
        }
        show
        eval [$page convert $itk_component(info)]
        lappend helpFiles $page
    }
    ...
}
```

Figure 5 – Using HelpPage objects to represent various types of help pages