# PyTix: Wrappers for the Tix Widget Set

Mike Clarkson,
Internet Discovery

Ioi Lam,
Sun Microsystems

November 30, 2000

## 1 Introduction

Tk/Tcl has long been an integral part of Python. It provides a robust and platform independent windowing toolkit, that is available to Python programmers using the `Tkinter` module. Tkinter is not the only GUI for Python, but is however the most commonly used one.

The Tix (Tk Interface Extension) library provides a rich set of widgets. Although the standard Tk library has many useful widgets, they are far from complete. The Tix library provides most of the commonly needed widgets that are missing from standard Tk: FileSelectBox, ComboBox, Control (a.k.a. SpinBox) and an assortment of scroll-able widgets. Tix also includes many more widgets that are generally useful in a wide range of applications: NoteBook, FileEntry, PanedWindow, etc. Figure 2 shows all of the Tix widgets - there are more than 40 of them.

With all these new widgets, you can introduce new interaction techniques into applications, creating more useful and more intuitive user interfaces. You can design your application by choosing the most appropriate widgets to match the special needs of your application and users.

Tkinter is a thin object-oriented layer on top of Tcl/Tk. To use Tkinter, you don't need to write Tcl code, but you will need to consult the Tk documentation, and occasionally the Tcl documentation. Tkinter is a set of wrappers that implement the Tk widgets as Python classes. In addition, the internal module `_tkinter` provides a thread-safe manner for two languages to interact: Python and Tcl.

## 2 Tix Widget Set

Tix introduces over 40 widgets to the Tk/Tkinter repertoire:

**Balloon** A balloon that pops up over a widget to provide help. When the user moves the cursor inside a widget to which a Balloon widget has been bound, a small pop-up window with a descriptive message will be shown on the screen.

**ButtonBox** The ButtonBox widget creates a box of buttons, such as is commonly used for `Ok Cancel`.

**CheckList** The CheckList widget displays a list of items to be selected by the user. CheckList acts similarly to the Tk checkbutton or radiobutton widgets, except it is capable of handling many more items than checkbuttons or radiobuttons.

**ComboBox** The Tix ComboBox widget is similar to the combo box control in MS Windows. The user can select a choice by either typing in the entry subwdget or selecting from the listbox subwidget.

**Control** The Control widget is also known as the `SpinBox` widget. The user can adjust the value by pressing the two arrow buttons or by entering the value directly into the entry. The new value will be checked against the user-defined upper and lower limits.

**DirSelectDialog** The DirSelectDialog widget presents the directories in the file system in a dialog window. The user can use this

dialog window to navigate through the file system to select the desired directory.

**DirList** The DirList widget displays a list view of a directory, its previous directories and its sub-directories. The user can choose one of the directories displayed in the list or change to another directory.

**DirTree** The DirTree widget displays a tree view of a directory, its previous directories and its sub-directories. The user can choose one of the directories displayed in the list or change to another directory.

**ExFileSelectBox** The ExFileSelectBox widget is usually embedded in a tixExFileSelectDialog widget. It provides an convenient method for the user to select files. The style of the ExFileSelectBox widget is very similar to the standard file dialog in MS Windows 3.1.

**FileSelectBox** The FileSelectBox is similar to the standard Motif(TM) file-selection box. It is generally used for the user to choose a file. FileSelectBox stores the files mostly recently selected into a ComboBox widget so that they can be quickly selected again.

**FileEntry** The FileEntry widget can be used to input a filename. The user can type in the filename manually. Alternatively, the user can press the button widget that sits next to the entry, which will bring up a file selection dialog.

**HList** The HList widget can be used to display any data that have a hierarchical structure, for example, file system directory trees. The list entries are indented and connected by branch lines according to their places in the hierachy.

**InputOnly** The purpose of `TixInputOnly` widgets are to accept inputs from the user, which can be done with the `bind` command.

**LabelEntry** The LabelEntry widget packages an entry widget and a label into one mega widget. It can be used be used to simplify the creation of "entry-form" type of interface.

**LabelFrame** The LabelFrame widget packages a frame widget and a label into one mega widget. To create widgets inside a LabelFrame widget, one creates the new widgets relative to the `frame` subwidget and manage them inside the `frame` subwidget.

**ListNoteBook** The ListNoteBook widget is very similar to the TixNoteBook widget: it can be used to display many windows in a limited space using a notebook metaphor. The notebook is divided into a stack of pages (windows). At one time only one of these pages can be shown. The user can navigate through these pages by choosing the name of the desired page in the `hlist` subwidget.

**Meter** The Meter widget can be used to show the pregress of a background job which may take a long time to execute.

**NoteBook** The NoteBook widget can be used to display many windows in a limited space using a notebook metaphor. The notebook is divided into a stack of pages. At one time only one of these pages can be shown. The user can navigate through these pages by choosing the visual "tabs" at the top of the NoteBook widget.

**OptionMenu** The OptionMenu creates a menu button of options.

**PanedWindow** The PanedWindow widget allows the user to interactively manipulate the sizes of several panes. The panes can be arranged either vertically or horizontally.The user changes the sizes of the panes by dragging the resize handle between two panes.

**PopupMenu** The Tix PopupMenu widget can be used as a replacement of the `tk_popup` command. The advantage of the Tix PopupMenu widget is it requires less application code to manipulate.

**Select** The Select widget is a container of button subwidgets. It can be used to provide radio-box or check-box style of selection options for the user.

**StdButonBox** The StdButonBox widget is a group of Standard buttons for Motif-like dialog boxes.

**TList** The TList widget can be used to display data in a tabular format. The list entries of a TList widget are similar to the entries in the Tk listbox widget. The main differences are (1) the TList widget can display the list entries in a two dimensional format and (2) you can use graphical images as well as multiple colors and fonts for the list entries.

**Tree** The Tree widget can be used to display hierachical data in a tree form. The user can adjust the view of the tree by opening or closing parts of the tree.

In addition, Tix augments Tk by providing

**Form** a form geometry manager based on attachment rules for all Tk widgets.

**pixmap** to create color images from XPM files.

**compound** Compound image types can be used to create images that consists of multiple horizontal lines; each line is composed of a series of items (texts, bitmaps, images or spaces) arranged from left to right. For example, a compound image can be used to display a bitmap and a text string simutaneously in a Tk `button` widget.

**wm** an addition to the standard TK `wm` command for reparenting windows.

Some of these widgets are implemented by Tix in "C", such as the HList and Tree widgets, but most are compound widgets of existing Tk widgets. They are all created using the simple object oriented programming (OOP) framework for writing mega-widgets called the *Tix Intrinsics*.

# 3 Integrating Hybrid Applications

As we have seen, Tix provides a rich widget set for designing user interfaces, and a simple object oriented framework for extending the widget repertoire with mega-widgets.

The `Tkinter` module is extended by Tix under Python by the module `Tix.py`. The Tix widgets are represented by a class hierarchy in Python with proper inheritance of base classes.

We set up an attribute access function so that it is possible to access subwidgets in a standard fashion, using

```
w.ok['text'] = 'Hello'
```

rather than

```
w.subwidget('ok')['text'] = 'Hello'
```

when `w` is a StdButtonBox. We can even do `w.ok.invoke()` because `w.ok` is subclassed from the Button class if you go through the proper constructors. A complete list of the subwidgets and methods for Tix widgets is presented in Appendix A.

Using Tkinter as a set of wrappers to implement the Tk widgets as Python classes may be suitable for small applications, but for large applications with tens of thousands of lines of UI code, this approach has its drawbacks. The Python code is much more verbose than the Tcl code, and the code expansion from using Tk widgets from Python is not offset from any commensurate benefits. With Tix, you can create larger and richer user interfaces, and sometimes the amount of user interface code exceeeds the ammount of application code. This suggests the possibility of explicitly using hybrid prgramming, where the user interface component of an application is written directly in Tk/Tix and the application code is written in Python.

In the following two subsections we will examine the Tkinter module and its thread safety to see the consequences of explicity working in two languages.

### 3.1 Tkinter Module

Besides providing a set of wrappers to implement the Tk widgets as Python classes, the `Tkinter` module provides a number of internal methods to facilitate communication between Python and Tcl. The following methods are available from Python to interact with Tcl:

**call** Concatenate the arguments to be a Tcl expression, and evaluate the resulting expression in Tcl at the global level.

**eval, globaleval** Evaluate an expression in Tcl; evaluate at the global level.

**evalfile** Evaluate a file of expressions in Tcl at the global level.

**setvar, getvar, unsetvar** Get, set and unset variables in Tcl, and return the results as a string.

**getint, getdouble, getboolean** Get variables in Tcl and return the results as the indicated type.

**exprlong, exprdouble, exprboolean** Evaluate an `expr` and return the result as the given type. Also, **exprstring**.

**splitlist** Split a string into a Tcl list.

**mainloop, dooneevent** Enter the main loop of Tk, handling Tcl events, or just do one event.

**quit** Quit the Tcl interpreter. All widgets will be destroyed. This is the replacement for the Tcl `exit` command.

**interpaddr** Get the address of the Tcl intepreter.

**register** Return a newly created Tcl function. If this function is called, the Python function FUNC will be executed. An optional function SUBST can be given which will be applied to the arguments before FUNC.

**nametowidget** Return the Tkinter instance of a widget identified by its Tcl name NAME.

In hybrid programming, these commands are used to communicate between the UI Tcl code and the Python main program. Thus we are free to design our application with the user interface portion written in Tcl/Tk/Tix, and the application portion written in Python. For example, any existing Tcl/Tk/Tix program can be executed immediately in Python using `evalfile`:

```
import Tix
root = Tix.Tk()
root.tk.evalfile('/tmp/hello.tcl')
root.mainloop()
```

Communication between Tix and Python can be facilitated using the `register` command. For example, lets consider a simple `hello.tcl` which contains a simple Hello World program with two buttons, one to print "Hello" on the stdout and one to exit the application:

```
frame .frame -relief ridge -borderwidth 2
pack .frame -fill both -expand 1
label .frame.label -text "Hello, World"
pack .frame.label -fill x -expand 1
button .frame.hello -text "Hello" \
       -command [list puts "Hello from Tcl"]
pack .frame.hello -side top
button .frame.button -text "Exit" \
       -command [list destroy .]
pack .frame.button -side bottom
```

We can define a function in Python to print "Hi from Python".

```
def Hi():
    """Demo function."""
    print "Hi from Python"
```

We register our Python function `Hi`, and then use the `eval` method of Tk to configure the button to call this command, before calling `mainloop`:

```
command = root.register(Hi)
root.tk.eval('.frame.hello configure \
       -text Hi -command ' + command)
```

Of course, the most useful Tcl function is the `pyeval` procedure:

```
command = root.register(eval)
root.tk.eval('proc pyeval {arg} {return [' \
+ command + ' $arg]}')
```

With this procedure you can execute any Python statements from Tcl.

### 3.2 Thread Safety

As the source code comments indicate, the threading situation is complicated. Paraphrasing from `Python-2.0/Modules/_tkinter.c`:

"Tk is not yet thread-safe, so we need to use a lock around all uses of Tcl. Previously, the Python interpreter lock was used for this. However, this causes problems when other Python threads need to run while Tcl is blocked waiting for events. To solve this problem, a separate lock for Tcl is introduced. Holding it is incompatible with holding Python's interpreter lock. `_tkinter.c` uses four C macros manipulate both locks together.

"`ENTER_TCL` and `LEAVE_TCL` are brackets, just like `Py_BEGIN_ALLOW_THREADS` and `Py_END_ALLOW_THREADS`. They should be used whenever a call into Tcl is made that could call an event handler, or otherwise affect the state of a Tcl interpreter. These assume that the surrounding code has the Python interpreter lock; inside the brackets, the Python interpreter lock has been released and the lock for Tcl has been acquired.

"Sometimes, it is necessary to have both the Python lock and the Tcl lock. (For example, when transferring data from the Tcl interpreter result to a Python string object.) This can be done by using different macros to close the `ENTER_TCL` block: `ENTER_OVERLAP` reacquires the Python lock (and restores the thread state) but doesn't release the Tcl lock; `LEAVE_OVERLAP_TCL` releases the Tcl lock.

"By contrast, `ENTER_PYTHON` and `LEAVE_PYTHON` are used in Tcl event handlers when the handler needs to use Python. Such event handlers are entered while the lock for Tcl is held; the event handler presumably needs to use Python. `ENTER_PYTHON` releases the lock for Tcl and acquires the Python interpreter lock, restoring the appropriate thread state, and LEAVE_PYTHON releases the Python interpreter lock and re-acquires the lock for Tcl. It is okay for `ENTER_TCL`/`LEAVE_TCL` pairs to be contained inside the code between `ENTER_PYTHON` and `LEAVE_PYTHON`."

Because of this locking mechanism, hybrid applications are now thread-safe, provided that the Tk loaded by Python does not itself have threads enabled.

## 4 Conclusion

Using the Tix widget set provides a wide range of high-level widgets that allows a richer class of user interfaces for Python applications. The Tix Intrinsics provides a simple object oriented programming framework, that allows Tix/Tkinter to be extended using mega-widgets, while taking advantage of Tcl's compactness and efficiency.

At the same time, such a hybrid architecture provides a graceful evolution path for legacy Tk/Tcl applications. With minor changes, Tix/Tk applications can be run almost immediately using Tix/Tkinter. Then, using the techniques and procedures described above, legacy applications might evolve by taking advantage of areas where Python's strengths are the most important.

This approach has the beneficial side effect of uncoupling the UI development from the functional core, promoting a healthy seperation between the two. Although this hybrid approach means that a project team must have programmers capable of two different languages, this is not necessarily onerous for a medium or large sized project. We have long accepted the dual programming role with C with interpreted langauges, and this is a similar approach: use a hybrid mix of languages where each language has its strongest feature.

# A Python Class Methods

For reference, we catalogue the Tix widgets and their subwidgets, and give the Python names of the methods defined on them. In general the naming convention follows the same approach as used in the `Tkinter` module. For details of what each method does, see the Tix documentation on http://tix.sourceforge.net in /dist/current/man/

**class TixWidget(Widget)**
**Methods:**
set_silent(self, value)
subwidget(self, name)
subwidgets_all(self)
config_all(self, option, value)

**class TixSubWidget(TixWidget)**
**Methods:**
destroy(self)

**class DisplayStyle**
**Methods:**
delete(self)
config(self, cnf={}, **kw)

**class Balloon(TixWidget)**
**Subwidgets:** label(Label)
message(Message)
**Methods:**
bind_widget(self, widget, cnf={}, **kw)
unbind_widget(self, widget)

**class ButtonBox(TixWidget)**
**Subwidgets:** Subwidgets are buttons added dynamically.
**Methods:**
add(self, name, cnf={}, **kw)
invoke(self, name)

**class CheckList(TixWidget)**
**Subwidgets:** hlist(HList) hsb(Scrollbar) vsb(Scrollbar)
**Methods:**
getstatus(self, entrypath)
setstatus(self, entrypath, status)

**class ComboBox(TixWidget)**
**Subwidgets:** entry(Entry) arrow(Button) slistbox(ScrolledListBox) tick(Button) cross(Button)
**Methods:**
add_history(self, str)
append_history(self, str)
insert(self, index, str)
pick(self, index)

**class Control(TixWidget)**
**Subwidgets:** incr(Button) decr(Button) entry(Entry) label(Label)
**Methods:**
decrement(self)
increment(self)
invoke(self)
update(self)

**class DirList(TixWidget)**
**Subwidgets:** hlist(HList) hsb(Scrollbar) vsb(Scrollbar)
**Methods:**
chdir(self, dir)

**class DirTree(TixWidget)**
**Subwidgets:** hlist(HList) hsb(Scrollbar) vsb(Scrollbar)
**Methods:**
chdir(self, dir)

**class ExFileSelectBox(TixWidget)**
**Subwidgets:** cancel(Button) ok(Button) hidden(Checkbutton) types(ComboBox) dir(ComboBox) file(ComboBox) dirlist(ScrolledListBox) filelist(ScrolledListBox)
**Methods:**
filter(self)
invoke(self)

**class ExFileSelectDialog(TixWidget)**
**Subwidgets:** fsbox(ExFileSelectBox)
**Methods:**
popup(self)
popdown(self)

**class FileSelectBox(TixWidget)**
**Subwidgets:** selection(ComboBox) filter(ComboBox) dirlist(ScrolledListBox) filelist(ScrolledListBox)

**Methods:**
apply_filter(self)
invoke(self)

**class FileSelectDialog(TixWidget)**
**Subwidgets:** btns(StdButtonBox)
fsbox(FileSelectBox)
**Methods:**
popup(self)
popdown(self)

**class FileEntry(TixWidget)**
**Subwidgets:** button(Button)
entry(Entry)
**Methods:**
invoke(self)
file_dialog(self)

**class HList(TixWidget)**
**Subwidgets:** None
**Methods:**
add(self, entry, cnf={}, **kw)
add_child(self, parent=None, cnf={}, **kw)
anchor_set(self, entry)
anchor_clear(self)
column_width(self, col=0, width=None, chars=None)
delete_all(self)
delete_entry(self, entry)
delete_offsprings(self, entry)
delete_siblings(self, entry)
header_create(self, col, cnf={}, **kw)
header_configure(self, col, cnf={}, **kw)
header_cget(self, col, opt)
header_exists(self, col)
header_delete(self, col)
header_size(self, col)
hide_entry(self, entry)
indicator_create(self, entry, cnf={}, **kw)
indicator_configure(self, entry, cnf={}, **kw)
indicator_cget(self, entry, opt)
indicator_exists(self, entry)
indicator_delete(self, entry)
indicator_size(self, entry)
info_anchor(self)
info_children(self, entry=None)
info_data(self, entry)
info_exists(self, entry)
info_hidden(self, entry)
info_next(self, entry)
info_parent(self, entry)
info_prev(self, entry)
info_selection(self)
item_cget(self, col, opt)
item_configure(self, entry, col, cnf={}, **kw)
item_create(self, entry, col, cnf={}, **kw)
item_exists(self, entry, col)
item_delete(self, entry, col)
nearest(self, y)
see(self, entry)
selection_clear(self, cnf={}, **kw)
selection_includes(self, entry)
selectiom_set(self, cnf={}, **kw)
show_entry(self, entry)
xview(self, *args)
yview(self, *args)

**class InputOnly(TixWidget)**
**Subwidgets:** None
**Methods:**

**class LabelEntry(TixWidget)**
**Subwidgets:** label(Label) entry(Entry)
**Methods:**

**class LabelFrame(TixWidget)**
**Subwidgets:** label(Label) frame(Frame)
**Methods:**

**class NoteBook(TixWidget)**
**Subwidgets:** nbframe(NoteBookFrame)
**Methods:**
add(self, name, cnf={}, **kw)
delete(self, name)
page(self, name)
pages(self)
raise_page(self, name)
raised(self)

**class NoteBookFrame(TixWidget)**
**Subwidgets:** None
**Methods:**
None: used to configure options that can be used to control the appearance of the TixNoteBook widget.

**class OptionMenu(TixWidget)**
**Subwidgets:** menubutton(Menubutton)
menu(Menu)
**Methods:**
add_command(self, name, cnf={}, **kw)
add_separator(self, name, cnf={}, **kw)
delete(self, name)
disable(self, name)
enable(self, name)

**class PanedWindow(TixWidget)**
**Subwidgets:** add(self, name, cnf={},
**kw)
panes(self)

**class PopupMenu(TixWidget)**
**Subwidgets:** menubutton(Menubutton)
menu(Menu)
**Methods:**
bind_widget(self, widget)
unbind_widget(self, widget)
post_widget(self, widget, x, y)

**class Select(TixWidget)**
**Subwidgets:** Subwidgets are buttons
added dynamically.
**Methods:**
add(self, name, cnf={}, **kw)
invoke(self, name)

**class StdButtonBox(TixWidget)**
**Subwidgets:** ok(Button) apply(Button)
cancel(Button) help(Button)
**Methods:**
invoke(self, name)

**class Tree(TixWidget)**
**Subwidgets:** None.
**Methods:**
autosetmode(self)
close(self, entrypath)
getmode(self, entrypath)
open(self, entrypath)
setmode(self, entrypath, mode='none')

Tix Class Structure

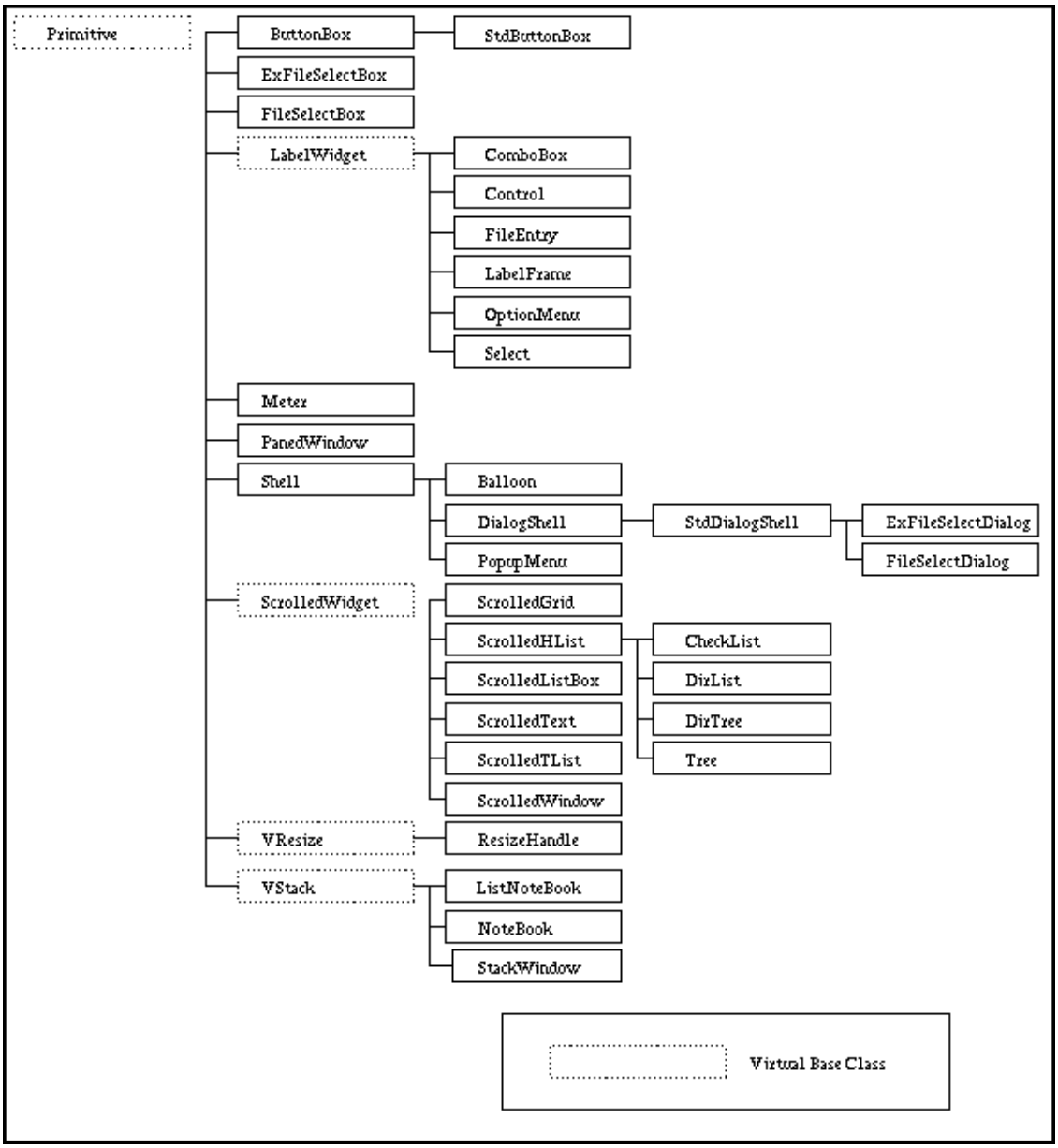


Figure 1: The Class Hierarchy of Tix Widgets