The Tix Widget Set

Tix Group,
http://tix.sourceforge.net

November 29, 2001.

Contents

1 Introduction

The Tix (Tk Interface Extension) library provides an additional rich set of widgets to Tk/Tkinter. 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 scrollable 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. In section 2, we review all of the widgets added by Tix. In section 3, we look at the simple object oriented class structure used to create the widgets added by Tix.

1.1 Installing Tix

To install Tix, consult the documentation in the docs directory in the Tix source distribution. You will want to look at:

- The Tix Installation Guide
- The Release Notes

To build Tix from source, you will require an installation of Tcl and Tk; see the ActiveState Tcl Home Page.

1.2 Using Tix with Python

Classes in the Tix module subclass the classes in the `Tkinter` module. The former imports the latter, so to use Tix with Tkinter, all you need to do is to import one module. In general, you can just import Tix, and replace the toplevel call to `Tkinter.Tk` with `Tix.Tk`:

```
import Tix
from Tkconstants import *
root = Tix.Tk()
```

To use Tix, you must have the Tix widgets installed, usually alongside your installation of the Tk widgets. To test your installation, try the following:

```
import Tix
root = Tix.Tk()
root.tk.eval('package require Tix')
```

If this fails, you have a Tk installation problem which must be resolved before proceeding. Use the environment variable `TIX_LIBRARY` to point to the installed Tix library directory, and make sure you have the dynamic object library ('tix8183.dll' or 'libtix8183.so') in the same directory that contains your Tk dynamic object library ('tk8183.dll' or 'libtk8183.so'). The directory with the dynamic object library should also have a file called 'pkgIndex.tcl' (case sensitive), which contains the line:

```
package ifneeded Tix 8.1 \
   [list load "[file join $dir tix8183.dll]" Tix]
```

2 Tix Widget Set

2.1 Tix Widgets

Tix introduces over 40 widget classes to the `Tk`/`Tkinter` repertoire. In the Tix distribution, there is a demo of all the Tix widgets in the 'demos' directory; in the Python standard distribution they are in the 'Demo/tix' directory.

For a more detailed description of the widgets, consult the manual pages.

2.1.1 Basic Widgets

tixBalloon a balloon that pops up over a widget to provide help. The Balloon widget can be used to show popped-up messages that describe the functions of the widgets in an application. 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.

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

tixComboBox 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.

tixControl 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.

tixLabelEntry 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. In this kind of interface, one must create many entry widgets with label widgets next to them and describe the use of each of the entry widgets.

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

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

tixOptionMenu The OptionMenu creates a menu button of options.

tixPopupMenu The Tix PopupMenu widget can be used as a replacement of the `tk_popup` command.

tixSelect 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.

tixStdButonBox The StdButonBox widget is a group of standard buttons for Motif-like dialog boxes.

2.1.2 File Selectors

tixDirList 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.

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

tixDirSelectDialog 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.

tixDirSelectBox The `tixDirSelectBox` is similar to the standard Motif(TM) directory-selection box. It is generally used for the user to choose a directory. DirSelectBox stores the directories mostly recently selected into a tixComboBox widget so that they can be quickly selected again.

tixExFileSelectBox 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.

tixFileSelectBox The FileSelectBox is similar to the standard Motif 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.

tixFileEntry 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 of the type specified by the `-dialogtype` option.

2.1.3 Hierachical ListBox

tixHList 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. The entries support images and text, to display with icons.

tixCheckList 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.

tixTree 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.

2.1.4 Tabular ListBox

tixTList 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.

2.1.5 Manager Widgets

tixPanedWindow The PanedWindow widget allows the user to interactively manipulate the sizes of several panes. The panes can be arranged either vertically or horizontally. Each individual pane may have upper and lower limits of its size. The user changes the sizes of the panes by dragging the resize handle between two panes.

tixNoteBook 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 (windows). 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.

tixListNoteBook 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. 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.

2.1.6 Image Types

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(n)` widget.

pixmap to create color images from XPM files.

2.1.7 Miscellaneous Widgets

InputOnly The InputOnly widget is to accept inputs from the user, which can be done with the `bind` command (UNIX only).

2.1.8 Form Geometry Manager

In addition, `Tix` augments Tk by providing:

tixForm Tix adds a form geometry manager based on attachment rules.

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 in fact, very few new widgets at the "C" level are introduced by Tix; 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.

2.2 Tix Commands

The tix commands provide access to miscellaneous elements of Tix's internal state and the Tix application context. Most of the information manipulated by these methods pertains to the application as a whole, or to a screen or display, rather than to a particular window.

To view the current settings, the common usage is in Tcl:

```
tix configure
```

or in Python

```
import Tix
root = Tix.Tk()
print root.tix_configure()
```

Method: `tix configure` *?option*
Query or modify the configuration options of the Tix application context. If no option is specified, returns a list (or dictionary in Python) of all of the available options. If option is specified with no value, then the method returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no option is specified). If one or more option-value pairs are specified, then the method modifies the given option(s) to have the given value(s); in this case the method returns an empty string. Option may be any of the configuration options.
Method: `tix cget` *option*
Returns the current value of the configuration option given by *option*. Option may be any of the configuration options.
Method: `tix getbitmap` *name*
Locates a bitmap file of the name `name.xpm` or `name` in one of the bitmap directories (see the `tix addbitmapdir` method). By using `tix getbitmap`, you can avoid hard coding the pathnames of the bitmap files in your application. When successful, it returns the complete pathname of the bitmap file, prefixed with the character '`@`'. The returned value can be used to configure the `bitmap` option of the Tk and Tix widgets.
Method: `tix addbitmapdir` *directory*
Tix maintains a list of directories under which the `tix getimage` and `tix getbitmap` methods will search for image files. The standard bitmap directory is '`$TIX_LIBRARY/bitmaps`'. The `tix addbitmapdir` method adds *directory*

into this list. By using this method, the image files of an applications can also be located using the `tix getimage` or `tix getbitmap` method.

Method: `tix filedialog` *?dlgclass*
Returns the file selection dialog that may be shared among different calls from this application. This method will create a file selection dialog widget when it is called the first time. This dialog will be returned by all subsequent calls to `tix filedialog`. An optional dlgclass parameter can be passed as a string to specified what type of file selection dialog widget is desired. Possible options are `tix`, `tixFileSelectDialog` or `tixExFileSelectDialog`.

Method: `tix getimage` *name*
Locates an image file of the name 'name.xpm', 'name.xbm' or 'name.ppm' in one of the bitmap directories (see the `tix addbitmapdir` method above). If more than one file with the same name (but different extensions) exist, then the image type is chosen according to the depth of the X display: xbm images are chosen on monochrome displays and color images are chosen on color displays. By using `tix getimage`, you can avoid hard coding the pathnames of the image files in your application. When successful, this method returns the name of the newly created image, which can be used to configure the `image` option of the Tk and Tix widgets.

Method: `tix option get` *name*
Gets the options maintained by the Tix scheme mechanism.

Method: `tix resetoptions` *newScheme newFontSet ?newScmPrio*
Resets the scheme and fontset of the Tix application to *newScheme* and *newFontSet*, respectively. This affects only those widgets created after this call. Therefore, it is best to call the resetoptions method before the creation of any widgets in a Tix application.

The optional parameter *newScmPrio* can be given to reset the priority level of the Tk options set by the Tix schemes.

Because of the way Tk handles the X option database, after Tix has been has imported and inited, it is not possible to reset the color schemes and font sets using the `tix configure` method. Instead, the `tix resetoptions` method must be used.

3 Tix Object Oriented Programming

Tix comes with a simple object oriented programming (OOP) framework, the Tix Intrinsics, for writing mega-widgets. The Tix Intrinsics is not a general purpose OOP system and it does not support some features found in general purpose OOP systems such as [incr Tcl] or Python. However, provides a simple and efficient interface for creating mega-widgets so that you can avoid the complexity and overheads of the general purpose OOP extensions.

The hard thing about programming with mega-widgets is to make sure that each instance you create can handle its own activities. Events must be directed to the right widget, procedures must act on data that is internal to that widget, and users should be able to change the options associated with the widget. For instance, we'll show an arrow widget that needs to know what direction it's pointing; this requires each instance of the widget to have its own variable. Furthermore, each widget must respond properly to changes requested by the application programmer during the program's run.

Arrow Buttons

3.1 Widget Classes and Widget Instances

All the mega-widget classes in Tix, such as TixComboBox and TixControl, are implemented in the Tix Intrinsics framework. You can write new widget classes with the Tix Intrinsics. In the next section, We'll go through all the steps of creating a new widget class in Tix. We'll illustrate the idea using a new class "TixArrowButton" as an example. TixArrowButton is essentially a button that can display an arrow in one of the four directions.

In this section we will use Tcl syntax, as the procedures described in this section are currently not exposed to the Python programmer using the Tix module.

3.1.1 Widget Instances

Each widget instance is composed of three integral parts: variables, methods and component widgets.

Variables Each widget instance is associated with a set of variables. In the example of an instance of the TixArrowButton class, we may use a variable to store the direction to which the arrow is pointing to. We may also use a variable to count how many times the user has pressed the button.

Each variable can be public or private. Public variables may be accessed by the application programmer (usually via `configure` or `cget` methods) and their names usually start with a dash (-). They usually are used to represent some user-configurable options of the widget instance. Private variables, on the other hand, cannot be accessed by the application programmer. They are usually used to store information about the widget instance that are of interest only to the widget writer.

All the variables of an instance are stored in a global array (dictionary) that has the same name as the instance. For example, the variables of the instance `.up` are stored in the global array `.up`. The public variable `-direction`, which records the direction to which the arrow is pointing to, is stored in `.up(-direction)`. The private variable `count`, which counts how many times the user has pressed the button, is stored in `.up(count)`. In comparison, the same variables of the `.down` instance are stored in `.down(-direction)` and `.down(count)`.

Methods To carry out operations on the widget, you define a set of methods. Each method can be declared as public or private. Public methods can be called by the application programmer. For example, if the `TixArrowButton` class supports the public methods `invoke` and invert, the application programmer can issue the commands to call these method for the widget instance `.up`.

```
.up invert
.up invoke
```

In contrast, private methods are of interests only to widget writers and cannot be called by application programmers.

Component Widgets A Tix mega-widget is composed of one or more component widgets. The main part of a mega-widget is called the root widget, which is usually a frame widget that encompasses all other component widgets. The other component widgets are called subwidgets.

The root widget has the same name as the mega-widget itself. In the above example, we have a mega-widget called `.up`. It has a root widget which is a frame widget and is also called `.up`. Inside `.up` we have a button subwidget called `.up.button`.

Similar to variables and methods, component widgets are also classified into public and private component widgets. Only public widgets may be accessed by the application programmer, via the `subwidget` method of each widget instance.

3.2 Widget Class Declaration

The first step of writing a new widget class is to decide the base class from which the new class. Usually, if the new class does not share any common features with other classes, it should be derived from the `TixPrimitive` class. If it does share common features with other classes, then it should be derived from the appropriate base class. For example, if the new class support scrollbars, it should be derived from `TixScrolledWidget`; if it displays a label next to its “main area", then it should be derived from `TixLabelWidget`.

In the case of our new `TixArrowButton` class, it doesn’t really share any common features with other classes, so we decide to use the base class `TixPrimitive` as its superclass.

3.2.1 Using the tixWidgetClass Command

We can use the `tixWidgetClass` command to declare a new class. The syntax is:

```
tixWidgetClass classCommandName {
    -switch value
    -switch value
    ....
}
```

For example, the following is the declaration section of `TixArrowButton`:

```
tixWidgetClass tixArrowButton {
    -classname  TixArrowButton
    -superclass tixPrimitive
    -method {
        flash invoke invert
    }
    -flag {
        -direction -state
    }
    -configspec {
        {-direction direction Direction e}
        {-state state State normal}
    }
    -alias {
        {-dir -direction}
    }
    -default {
        {*Button.anchor         c}
        {*Button.padX           5}
    }
}
```

We'll look at what each option means as the command is described in the following sections. The first argument for `tixWidgetClass` is the command name for the widget class (`tixArrowButton`). Command names are used to create widgets of this class. For example, the code:

```
tixArrowButton .arrow
```

creates a widget instance `.arrow` of the class `TixArrowButton`. Also, the command name is used as a prefix of all the methods of this class. For example, the `Foo` and `Bar` methods of the class `TixArrowButton` will be written as `tixArrowButton:Foo` and `tixArrowButton:Bar`.

The class name of the class (`TixArrowButton`) is specified by the `-classname` switch inside the main body of the declaration. The class name is used only to specify options in the TK option database. Notice the difference in the capitalization of the class name and the command name of the `TixArrowButton` class: both of them have the individual words capitalized, but the command name (`tixArrowButton`) starts with a lower case letter while the class name (`TixArrowButton`) starts with an upper case letter. When you create your own classes, you should follow this naming convention.

The `-superclass` switch specifies the superclass of the new widget. In our example, we have set it to `tixPrimitive`.

3.3 Writing Methods

After we have declared the new widget class, we can write methods for this class to define its behavior. Methods are just a special type of TCL procedures and they are created by the `proc` command. There are, however, two requirements for methods. First, their names must be prefixed by the command name of their class. Second, they must accept at least one argument and the first argument that they accept must be called `w`.

For example, the following is an implementation of the invert method for the class `TixArrowButton`:

```
proc tixArrowButton:invert {w} {
    upvar #0 $w data

    set curDirection $data(-direction)
    case $curDirection {
        n {
            set newDirection s
        }
        s {
            set newDirection n
        }
        # ....
    }
}
```

Notice that the name of the method is prefixed by the command name of the class (tixArrowButton). Also, the first and only argument that it accepts is `w` and the first line it executes is `upvar #0 $w data`.

The argument `w` specifies which widget instance this method should act upon. The `invert` method is used to invert the direction of the arrow. Therefore, it should examine the variable `.up(-direction)`, which stores the current direction of the instance `.up`, and modify it appropriately. The `upvar #0 $w data` tells the intepreter that the array data should be an alias for the global array whose name is stored in `$w`. We will soon see how the widget's methods use the data array.

3.3.1 Declaring Public Methods

All the methods of a class are by default private methods and cannot be accessed by the application programmer. If you want to make a method public, you can include its name in the `-method` section of the class declaration. In our `TixArrowButton` example, we have declared that the methods `flash`, `invert` and `invoke` are public methods and they can be accessed by the application programmer. All other methods of the `TixArrowButton` class will be private.

3.4 Standard Initialization Methods

Each new mega-widget class must supply three standard initialization methods. When an instance of a Tix widget is created, three methods will be called to initialize this instance. The methods are `InitWidgetRec`, `ConstructWidget` and `SetBindings` and they will be called in that order. The following sections show how these methods can be implemented.

3.4.1 The InitWidgetRec Method

The purpose of the `InitWidgetRec` method is to initialize the variables of the widget instance. For example, the following implementation of `tixArrowButton:InitWidgetRec` sets the count variable of each newly created instance to zero.

```
proc tixArrowButton:InitWidgetRec {w} {
    upvar #0 $w data

    set data(count) 0
}
```

Chaining Methods

The above implementation is not sufficient because our `TixArrowButton` class is derived from `TixPrimitive`. The class derivation in Tix is basically an is-a relationship: `TixArrowButton` is a `TixPrimitive`. `TixPrimitive` defines the method `tixPrimitive:InitWidgetRec` which sets up the instance variables of every instance of `TixPrimitive`. Since an instance of `TixArrowButton` is also an instance of `TixPrimitive`, we need to make

sure that the instance variables defined by `TixPrimitive` are also properly initialized. The technique of calling a method defined in a superclass is called the chaining of a method.

The `tixChainMethod` command will automatically find a superclass that defines the method we want to chain and calls this method for us. For example:

```
proc tixArrowButton:InitWidgetRec {w} {
    upvar #0 $w data

    tixChainMethod $w InitWidgetRec
    set data(count) 0
}
```

Notice the order of the arguments for `tixChainMethod`: the name of the instance, `$w`, is passed before the method we want to chain, `InitWidgetRec`.

3.4.2 The ConstructWidget Method

The `ConstructWidget` method is used to create the components of a widget instance. In the case of `TixArrowButton`, we want to create a new button subwidget, whose name is `button`, and use a bitmap to display an arrow on this button. Assuming the bitmap files are stored in the files up.xbm, down.xbm, left.xbm and right.xbm, the string substitution `@$data(-direction).xbm` will give us the appropriate bitmap depending on the current direction option of the widget instance.

```
proc tixArrowButton:ConstructWidget {w} {
    upvar #0 $w data

    tixChainMethod $w ConstructWidget

    set data(w:button) \
        [button $w.button \
        -bitmap @$data(-direction).xbm]
    pack $data(w:button) \
        -expand yes -fill both
}
```

The `tixArrowButton:ConstructWidget` method shown above sets the variable `data(w:button)` to be the pathname of the button subwidget.

As a convention of the Tix Intrinsics, we must declare a public subwidget `swid` by storing its pathname in the variable `data(w:swid)`.

3.4.3 The SetBindings Method

In your interface, you want to handle a lot of events in the subwidgets that make up your mega-widget. For instance, when somebody presses the button in a `TixArrowButton` widget, you want the button to handle the event. The `SetBindings` method is used to creates event bindings for the components inside the mega-widget. In our TixArrowButton example, we use the bind command to specify that the method `tixArrowButton:IncrCount` should be called each time when the user presses the first mouse button. As a result, we can count the number of times the user has pressed on the button.

```
proc tixArrowButton:SetBindings {w} {
    upvar #0 $w data

    tixChainMethod $w SetBindings

    bind $data(w:button) <1> \
        "tixArrowButton:IncrCount $w"
}

proc tixArrowButton:IncrCount {w} {
    upvar #0 $w data

    incr data(count)
}
```

3.5 Declaring and Using Variables

The private variables of a widget class do not need to be declared. In fact they can be initialized and used anywhere by any method. Usually, however, general purpose private variables are initialized by the `InitWidgetRec` method and subwidget variables are initialized in the `ConstructWidget` method.

We have seen in the `tixArrowButton:InitWidgetRec` example that the private variable `data(count)` was initialized there. Also, the private variable `data(w:button)` was initialized in `tixArrowButton:ConstructWidget`

and subsequently used in `tixArrowButton:SetBindings`. In contrast, public variables must be declared inside the class declaration. The following arguments are used to declare the public variables and specify various options for them:

-flag As shown in the class declaration in the figure, the -flag argument declares all the public variables of the `TixArrowButton` class to be -direction and -state.

-configspec We can use the -configspec argument to specify the details of each public variable. For example, the following declaration:

```
-configspec {
    {-direction direction Direction e}
    {-state state State normal}
}
```

specifies that the -direction variable has the resource name direction and resource class Direction; its default value is `e`. The application programmer can assign value to this variable by using the -direction option in the command line. The declaration of -state installs similar definitions for that variable.

3.5.1 Initialization of Public Variables

When a widget instance is created, all of its public variables are initialized by the Tix Intrinsics before the `InitWidgetRec` method is called. Therefore, `InitWidgetRec` and any other method of this widget instance are free to assume that all the public variables have been properly initialized and use them as such.

The public variables are initialized by the following criteria:

Step 1 If the value of the variable is specified by the creation command, this value is used. For example, if the application programmer has created an instance in the following way:

```
tixArrowButton .arr -direction n
```

The value `n` will be used for the -direction variable.

Step 2 if step 1 fails but the value of the variable is specified in the options database, that value is used. For example, if the user has created an instance in the following way:

```
option add *TixArrowButton.direction w
tixArrowButton .arr
```

The value `w` will be used for the -direction variable.

Step3 if step 2 also fails, the default value specified in the -configspec section of the class declaration will be used.

You can use a type ckecker procedure to check whether the user has supplied a value of the correct type for a public variable. The type checker is specified in the -configspec section of the class declaration after the default value.

3.5.2 Public Variable Configuration Methods

After a widget instance is created, the user can assign new values to the public variables using the configure method. For example, the following code changes the -direction variable of the `.arr` instance to `n`.

```
.arr configure -direction n
```

In order for configuration to work, you have to define a configuration method that does what the programmer expects. The configuration method of a public variable is invoked whenever the user calls the configure method to change the value of this variable. The name of a configuration method must be the name of the public variable prefixed by the creation command of the class and :config. For example, the name configuration method for the -direction variable of the `TixArrowButton` class is `tixArrowButton:config-direction`. The following code implements this method:

```
proc tixArrowButton:config-direction \
        {w value} {
    upvar #0 $w data

    $data(w:button) config \
        -bitmap @$value.xbm
}
```

Notice that when `tixArrowButton:config-direction` is called, the value parameter contains the new value of the -direction variable but `data(-direction)` contains the old value. This is useful when the configuration method needs to check the previous value of the variable before taking in the new value.

If a type checker is defined for a variable, it will be called before the configuration method is called. Therefore, the configuration method can assume that the type of the value parameter is got is always correct.

If you do not need to override the value, you don't need to return anything from the configuration method. In this case, the Tix Intrinsics will assign the new value to the instance variable for you.

For efficiency reasons, the configuration methods are not called during the intialization of the public variables. If you want to force the configuration method to be called for a particular public variable, you can specify it in the `-forcecall` section of the class declaration. In the following example, we force the configuration method of the -direction variable to be called during intialization:

```
-forcecall {
    -direction
}
```

4 Using Tix with Python

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.

In our example from the previous section, we would make our new mega-widget available to Python by extending the Tix module with the following class:

```
class ArrowButton(TixWidget):
  """ArrowButton - Demo Compound Widget.
  Subwidget Class
  ------ ---
  button    Button
  """
  def __init__(self, master, cnf={}, **kw):
      TixWidget.__init__(self, master,
               'tixArrowButton',
               ['options'], cnf, kw)
      self.subwidget_list['button'] =
              _dummyButton(self, 'button')
  def flash(self):
      self.tk.call(self._w, 'flash')
  def invert(self):
      self.tk.call(self._w, 'invert')
  def invoke(self):
      self.tk.call(self._w, 'invoke')
```

4.1 Freezing Tix Programs

Freeze (`$PYTHONHOME/tools/freeze/freeze.py`) make it possible to ship arbitrary Python programs to people who don't have Python. The shipped file (called a "frozen" version of your Python program) is an executable, so this only works if your platform is compatible with that on the receiving end. The shipped file contains a Python interpreter and large portions of the Python runtime. Some measures have been taken to avoid linking unneeded modules, but the resulting binary is usually not small.

The Python source code of your program (and of the library modules written in Python that it uses) is not included in the binary – instead, the compiled byte-code is incorporated. This gives some protection of your Python source code, though not much – a disassembler for Python byte-code is available in the standard Python library. At least someone running "strings" on your binary won't see the source.

With Python 2.x, is is possible to freeze Tix programs under Unix and Windows. Currently you must also deliver your frozen program with a set of Tcl/Tk/Tix library files. The best way to ship a frozen Tkinter program is to decide in advance where you are going to place the Tcl/Tk/Tix library files in the distributed setup, and then declare these directories in your frozen Python program using the TCL_LIBRARY, TK_LIBRARY and TIX_LIBRARY environment variables.

For example, assume you will ship your frozen program in the directory `<root>/bin/windows-x86` and will place your Tcl/Tk/Tix library files in `<root>/lib/tcl8.3` in `<root>/lib/tk8.3` and `<root>/lib/tix8.1` respectively. Then placing the following lines in your frozen Python script before importing Tkinter or Tix would set the environment correctly for Tcl/Tk/Tix:

```
import sys, os, os.path
Parent = os.path.dirname(os.getcwd())
RootDir = os.path.dirname(Parent)

if os.name == "nt":
  sys.path = [", '..\\..\\lib\\python-2.2']
  lib = RootDir + '\\lib\\'
  os.environ['TCL_LIBRARY'] = lib + 'tcl8.3'
  os.environ['TK_LIBRARY'] = lib + 'tk8.3'
  os.environ['TIX_LIBRARY'] = lib + 'tix8.1'
elif os.name == "posix":
  sys.path = [", '../../lib/python-2.2']
  lib = RootDir + '/lib/'
```

```
os.environ['TCL_LIBRARY'] = lib + 'tcl8.3'
os.environ['TK_LIBRARY'] = lib + 'tk8.3'
os.environ['TIX_LIBRARY'] = lib + 'tix8.1'
```

This also adds `<root>/lib/python-2.2` to your Python path for any Python files such as `_tkinter.pyd` you may need.

Note that the dynamic libraries (such as tcl83.dll tk83.dll python22.dll under Windows, or libtcl8.3.so and libtcl8.3.so under Unix) are required at program load time, and are searched by the operating system loader before Python can be started. Under Windows, the environment variable `PATH` is consulted, and under Unix, it may be the the environment variable `LD_LIBRARY_PATH` and/or the system shared library cache (ld.so). An additional preferred directory for finding the dynamic libraries is built into the .dll or .so files at compile time - see the `LIB_RUNTIME_DIR` variable in the Tcl makefile. The OS must find the dynamic libraries or your frozen program won't start. Usually we make sure that the .so or .dll files are in the same directory as the executable, but this may not be foolproof.

A workaround to installing your Tcl library files with your frozen executable would be possible, by freezing the Tcl/Tk/Tix code into the dynamic libraries using the Tix Stand-Alone-Module (SAM) module. This is currently untested, but the maintainers of Tix would welcome feedback on this point.

There are some caveats using frozen Tkinter applications:

- Under Windows if you use the -s windows option (recommended), writing to stdout or stderr is an error. This makes debugging very dificult. If possible, develop and freeze first under Unix, where you can debug to stdout. Then make sure the frozen application never writes to stdout or stderr and try freezing under Windows.

- The Tcl `[info nameofexecutable]` will be set to where the program was frozen, not where it is run from.

- The global variables argc and argv do not exist.

References

Web Pages

Tix http://tix.sourceforge.net

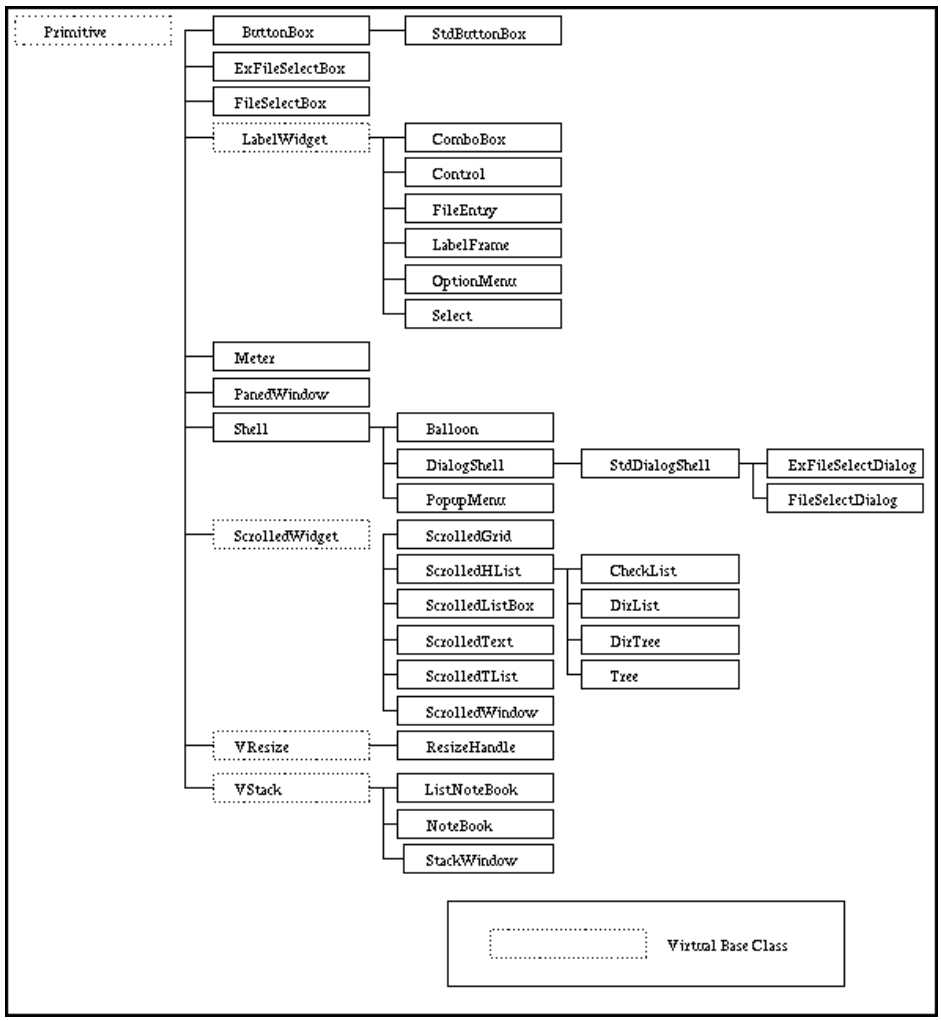
Python http://www.python.org

Tk/Tcl http://dev.scriptics.com

Tkinter http://www.python.org

Tkinter3000 http://tkinter.effbot.org

Tix Class Structure



The Class Hierarchy of Tix Widgets