The following paper was originally presented at the
Third Annual Tcl/Tk Workshop
sponsored by Unisys, Inc. and USENIX Association
Toronto, Ontario, Canada, July 1995.

[incr Widgets]
An Object Oriented Mega-Widget Set

Mark L. Ulferts
DSC Communications Corporation

[incr Widgets]
An Object Oriented Mega-Widget Set

Mark L. Ulferts

DSC Communications Corporation

Switching Products Division

mulferts@spd.dsccc.com

http://www.wn.com/biz/iwidgets

Abstract

The introduction of [incr Tcl] and [incr Tk] allows an object oriented approach to Tk widget construction. "Mega-widgets" developed in these extensions seamlessly expand the Tk base widget set. Each of these object-oriented widgets may themselves be extended, using either inheritance or composition. This paper presents one such general purpose hierarchy called [incr Widgets] which maintains the Motif look-and-feel and establishes several new concepts, including extensible child sites and flexible large scale component configuration.

Introduction

Typically, Tcl/Tk application development leads to the redundant creation of widget combination patterns which can be singled out for replacement with higher level abstractions. For example, a label is usually associated with an entry widget, listboxes frequently have attached scrollbars, and dialogs require buttons and modality. This is due to the simplicity of the Tk widget set. Seasoned developers commonly package this code, attempting to create a composite widget in a set of procedures which allows for consistent creation of the widget combination. This may provide centralization of logic, but the procedures lack the encapsulation of a pure widget and end up flooding global name space. At this point, some developers may resort to C code.

What was really needed was the ability to combine Tk widgets together into abstract building blocks called "Mega-Widgets" at the VHLL layer. The [incr Tcl] [1] and [incr Tk] [2] extensions provide this capability, allowing mega-widget development in an object-oriented paradigm using Tcl/Tk rather than C. The next step was to build a mega-widget set in these extensions which replaces the redundant widget combination patterns and provides a foundation for future development. [incr Widgets] is one such extension.

[incr Widgets] is an object-oriented, extensible set of mega-widgets, delivering many general purpose widgets such as option menus, selection boxes, and dialogs whose counterparts are found in Motif. Since [incr Widgets] is based on [incr Tk], the Tk framework of configuration options and widget commands is maintained. In other words, they look, act, and feel like Tk widgets. [incr Widgets] blends with the standard Tk widgets, raising the level of programming and making it easier to consistently develop well styled applications.

The idea of extending the basic Tk widget set is not original. Other mega-widget extensions exist such as Tix [3] and itcl-widgets [4]. Visually, [incr Widgets] covers some of the same ground, successfully replacing many of the same typical combinations. [incr Widgets] differs in the degree of its reusability, extensibility, flexibility, and adherence to the Motif style guide.

The [incr Widgets] mega-widget set is also distinguished by its consistent use of style, built-in intelligence, high degree of flexibility, ease of extending base level functionality, and its object-oriented implementation. Its use has resulted in increased productivity, reliability, and style guide adherence. This paper concentrates on these unique aspects of the widget set and the presentation of its innovative concepts. A pictorial tour with sample code segments will be given as an appendix.

Mega-Widgets

Mega-widgets has been a hot topic within the Tcl/Tk community. The discussion centers on the benefits, frameworks, mechanisms, and implementation techniques. It was McLennan [1] who originally coined the term in his work with [incr Tcl], expanding on the concept with [incr Tk]. He proposes that mega-widgets should seamlessly extend the Tk widget set. They should behave like standard Tk widgets, but are composed of many Tk widgets and possibly other mega-widgets as components. The implementation must ensure that users notice no significant differences. Standard commands such as ‘configure’ and ‘cget’ must exist and options should be propagated to all components. Thus, configuration of a mega-widgets “-background” or “-relief” option should have the expected results on its components.

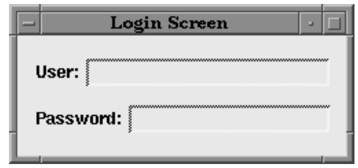
The [incr Tcl] and [incr Tk] extensions fully address the issues of framework and mechanism for mega-widget production. They have established themselves as the defacto standard object-oriented extensions and have been chosen to provide the backbone for the [incr Widgets] set. Using these extensions, each mega-widget becomes a “class”, defining a unique type of widget object in a separate namespace. This ensures that data and commands associated with an object are encapsulated, eliminating global name space pollution.

Example

At this point, an example of mega-widget usage can provide a taste of [incr Widgets] capabilities and illustrate the benefits. The example centers around the construction of a typical login screen which prompts the user for user name and password. New requirements will be incremental, leading to the development of a new mega-widget which is implemented as an extension of an existing one.

A primitive login screen is composed of two fields, each having a label and entry widget. The Tcl/Tk code required is shown in Figure 1. [incr Widgets] provides an EntryField class which replaces this standard widget combination. This is shown in Figure 2. At this point, mega-widget usage is mostly a matter of convenience and minor savings in code, yet with a few additional requirements the benefits begin to escalate.

Now let’s add new requirements which might be required for a normal login screen. First, the labels should be left aligned. Next, the user field width should be limited to a maximum of 10 characters with input restricted to alphabetic characters and illegal character entry ringing the bell. As for the password, input must be masked and the return key should invoke a login procedure. We’ll also present a more aesthetic interface by varying the textual background in the mega-widget. Since this last requirement applies to both EntryFields, the option database will be used. Figure 3 illustrates the code needed to implement these new requirements using




```
frame .login
pack .login -padx 10 -pady 10

frame .login.userFrame
label .login.userFrame.userLabel \
    -text "User:"
entry .login.userFrame.userEntry

pack .login.userFrame -fill x -pady 5
pack .login.userFrame.userLabel \
    -side left
pack .login.userFrame.userEntry \
    -side left -expand yes -fill x

frame .login.passwdFrame
label .login.passwdFrame.passwdLabel \
    -text "Password:"
entry .login.passwdFrame.passwdEntry

pack .login.passwdFrame -fill x -pady 5
pack .login.passwdFrame.passwdLabel \
    -side left
pack .login.passwdFrame.passwdEntry \
    -side left -expand yes -fill x
```

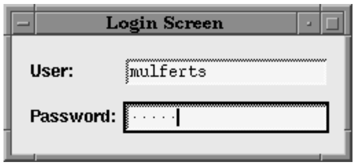
FIGURE 1 - Tcl/Tk Login screen

```
frame .login
pack .login -padx 10 -pady 10

EntryField .login.user \
    -labeltext "User:" -labelon yes
pack .login.user -fill x -pady 5

EntryField .login.passwd \
    -labeltext "Password:" -labelon yes
pack .login.passwd -fill x -pady 5
```

FIGURE 2 - [incr Widgets] Login screen

[incr Widgets]. Even without the presentation of comparative straight Tcl/Tk code, its safe to say that the benefits have increased.

This example gives just a sampling of the label control capabilities built into those classes based on the LabeledWidget class such as the EntryField mega-widget. The label’s position relative to its associated widget may be specified using standard directions: nw, n, ne, e, se, s, sw, and w. The label need not be limited to text, the class supports both bitmaps and images as well. A mar-



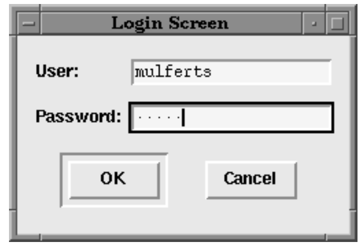

```
option add *textBackground “GhostWhite”

frame .login
EntryField .login.user -labeltext “User:” \
    -labelon yes -width 10 -fixed yes \
    -type alphabetic -invalid bell

EntryField .login.passwd
    -labeltext “Password:” -labelon yes \
    -type password -command LoginProc

LabeledWidget::alignLabels \
    .login.user .login.passwd

pack .login -padx 10 -pady 10
pack .login.user -fill x -pady 5
pack .login.passwd -fill x -pady 5
```

FIGURE 3 - Login screen with aligned labels

gin between the label and its associated widget may be given. Alignment is provided by adjusting the margins of a group of LabeledWidget based mega-widgets.

Currently, our login screen lacks a method of cancellation barring closure from the window manager decoration. Since this is not the most elegant method of window removal, “OK” and “Cancel” buttons seem like worthy additions. A well styled application would also make the buttons be of equal width and signify a default button associated with striking the return key through the appearance of an encompassing sunken ring. The ButtonBox class provides this functionality, making button management simple. As a manager widget, the ButtonBox controls the orientation, separation, and size of its button components. Buttons are added with the ‘add’ command. The ‘default’ command allows specification of a button within a sunken ring. Figure 4 presents the improved login screen.

Expanding further, a truly useful login screen should be a modal toplevel dialog widget. The [incr Widgets] Dialog class supports system, application[1], and non-modal dialogs. The difference being the degree of blocking. System modal dialogs block all applications, whereas application modal dialogs only block the current application. This allows processing of the dialog contents following user response and dialog termination. Non-modal dialogs are non-blocking, enabling the application to continue. In this case, the actions attached to the buttons should perform all processing of the dialog contents.




```
option add *textBackground “GhostWhite”

frame .login
EntryField .login.user -labeltext “User:” \
    -labelon yes -width 10 -fixed yes \
    -type alphabetic -invalid bell \
    -command {.login.bbox invoke}

EntryField .login.passwd \
    -labeltext “Password:” -labelon yes \
    -command {.login.bbox invoke} \
    -type password

LabeledWidget::alignLabels \
    .login.user .login.passwd

ButtonBox .login.bbox -orient horizontal
.login.bbox add -text OK -command LoginProc
.login.bbox add -text Cancel -command exit
.login.bbox default OK

pack .login -padx 10 -pady 10
pack .login.user -fill x -pady 5
pack .login.passwd -fill x -pady 5
pack .login.bbox -fill x
```

FIGURE 4 - Login screen with buttons

The Dialog mega-widget class also contains a predefined extensible location called a “child site”. This is an internally packed standard Tk frame which may be used as a parent for whole combinations of user specified widgets. Figure 5 illustrates the position of the child

FIGURE 5 - Dialog child site

1. Application modal dialogs are implemented through the use of the BLT [5] extension.

site frame in an instance of the Dialog class. In the login screen example, this frame can be filled with the user name and password EntryField mega-widgets.

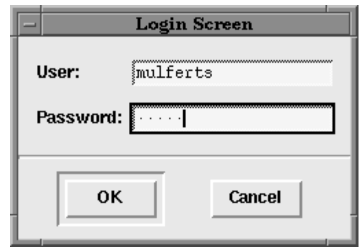
Once a dialog is created, it is displayed based on modality via the ‘activate’ command. For application and system modal dialogs, control is not immediately returned. Instead, it is delayed until invocation of the ‘deactivate’ command which accepts an optional argument that is returned as a result of the ‘activate’ command. This allows user control of dialog unmapping, status notification, and determination.

For example, two buttons could be added to a system modal dialog, each button specifying a command which executes the ‘deactivate’ command with a unique argument. The application could then activate the dialog, wait for deactivation, and perform actions based on the return value. This could all be placed in an “if” statement. The Dialog class uses this optional deactivation argument to provide default return values of zero and one for the “OK” and “Cancel” buttons as indicators of the dialog exit status. This ability proves useful for standard dialog management.

Figure 6 illustrates the new login screen implemented as an application modal Dialog composed of the two EntryFields. The need for explicit default button bindings has been left to the Dialog class, making the application even cleaner.The comparative amount of Tcl/Tk code required to provide the same flexible functionality would be quite substantial.

Since [incr Widgets] was designed to be a means rather than an end, each mega-widget is itself extensible. [incr Tk] provides the mechanism and framework to build new mega-widgets based upon existing ones using object-oriented techniques such as inheritance and composition. [incr Widgets] provides “child sites” which enable the visual aspects of a mega-widget to be augmented.

The login screen example could benefit from this capability. A new “Login” mega-widget derived from the Dialog class can be created, encapsulating the combination of widgets required to implement login screen functionality and enable reuse across many new projects. As a mega-widget, the Login class should maintain the standard options such as background and cursor. It should also provide unique options for specifying the labels of the entry widgets so they may be easily modified. Figure 7 shows the [incr Tcl]/[incr Tk] code needed to implement the “Login” mega-widget class.




```
option add *textBackground "GhostWhite"

Dialog .login -disphelp no \
    -dispapply no -modality application

set cs [.login childSite]

EntryField $cs.user -labeltext "User:" \
    -labelon yes -width 10 -fixed yes \
    -type alphabetic

EntryField $cs.passwd -labelon yes \
    -labeltext "Password:" -type password

pack $cs.user -fill x -pady 5
pack $cs.passwd -fill x -pady 5

LabeledWidget::alignLabels \
    $cs.user $cs.passwd

if {[.login activate]} {
  LoginProc [$cs.user get] [$cs.passwd get]
}
```

FIGURE 6 - Login dialog

The Login mega-widget can now be reused in new applications. It can be used as the front end to a database or a system administration tool. Since the labels were made public, the Login class can even be internationalized. For example, the “-userlabel”, “-passwdlabel” options could be given in a foreign dialect or read from a language specific configuration file. Since the Login class was derived from the Dialog class, the button labels may be modified as well. To illustrate, Figure 8 depicts an instance of the Login mega-widget in Spanish.

One final point. It should be noted that the lack of an option being made public does not make it inaccessible. The dilemma is that keeping all options tends to cause option explosion, yet only providing a few limits usefulness. As a general rule, standard options should be kept as well as frequently used options. In the Login mega-widget, standard options were kept and each label was provided a unique option due to a high degree of anticipated usage. Other options such as “-foreground” can be accessed on an as needed basis via the [incr Tk] ‘com-

```
itcl::class Login {
    inherit Dialog

    constructor {args} {
        Dialog::constructor -disphelp no -dispapply no
    } {
        itk_component user {
            EntryField $itk_hull.user -labeltext "User:" -labelon yes -width 10 \
                                 -fixed yes -type alphabetic
        } {
             keep -cursor -background
        }
        pack $itk_component(user) -fill x -pady 5

        itk_component passwd {
           EntryField $itk_hull.passwd -labeltext "Password:" -labelon yes -type password
        } {
           keep -cursor -background
        }
        pack $itk_component(passwd) -fill x -pady 5

        itk_option -userlabel userLabel Text "User:" {
           $itk_component(user) configure -labeltext $itk_option(-userlabel)
           LabeledWidget::alignLabels $itk_component(user) $itk_component(passwd)
        }

        itk_option -passwdlabel passwdLabel Text "Password:" {
           $itk_component(passwd) configure -labeltext $itk_option(-passwdlabel)
           LabeledWidget::alignLabels $itk_component(user) $itk_component(passwd)
        }

        eval configure $args
    }

    method name {} {
        return [$itk_component(user) get]
    }

    method passwd {} {
        return [$itk_component(passwd) get]
    }
}

Login .login -title "Login Screen" -modality application

if {[.login activate]} {
    LoginProc [.login name] [.login passwd]
}
```

FIGURE 7 - [incr Tcl]/[incr Tk] Login mega-widget class

```
Login .login -title "Spanish Login Screen" \
    -userlabel "Nombre:" \
    -passwdlabel "Contrasena:" \
    -cancellabel "Cancelar" -oklabel "Bien" \
    -modality application

if {[.login activate]} {
    LoginProc [.login name] [.login passwd]
}
```

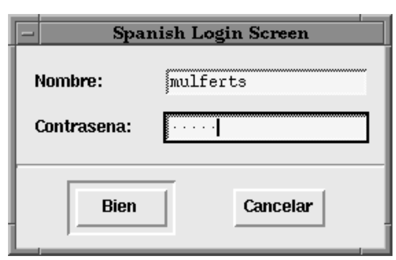


FIGURE 8 - Spanish login screen

ponent' command or using the option database. Figure 9 illustrates both of these methods of component access.

```
option add *Login*user.foreground Red

                 Or

.login component user configure \
       -foreground Red
```

FIGURE 9 -Login component access

Benefits

The benefits of mega-widget usage increase proportionally with the complexity of the application. This was readily apparent in the login screen example. Replacement of the more elemental patterns was mostly a matter of convenience. Yet as requirements were added, the code savings became substantial. As the example reached a medium level of complexity, the benefits extended to consistent usage of style. Productivity gains also became quite noticeable.

As applications increase in size, invariably requiring a main window and numerous dialogs, mega-widget usage offers significant productivity gains and increased reliability. This could also be seen in the example as well. A savings of a sizable amount of straight Tcl/Tk code was achieved and implementation of such things as modality was abstracted into the Dialog mega-widget and it's base classes. Since the Dialog class is encapsulated and tested, the Login mega-widget was built on a sound foundation. Errors typical of "cut and paste" built applications, such as forgetting to release a grab, have been eliminated. Developers are free to concentrate on the application and not low level problems.

To draw an analogy to current building construction techniques, Tk widgets are bricks and mega-widgets are pre-formed walls built with bricks. It is much quicker to construct a large building using walls than bricks. Although in the construction industry, this creates a lot of cookie cutter, identical, and boring buildings, application users appreciate this consistency, especially when it increases usability. A user shouldn't be confused during the operation of an application by being presented variant combinations of the same widget patterns. Each one having a unique behavior which users must learn during operation, rather than from prior experiences with other more standard interfaces.

The significance of a consistent style should not be overlooked. It doesn't always appear in typical Tcl/Tk applications. This stems from Tk itself. Its greatest asset is also a liability: a simple and easy to use widget set. Tk can make it easy for novice programmers to construct interfaces which conform to random personal styles rather than any known industry standards. Many unusual applications have been produced with Tk which have sunken buttons, raised entries, or are packed with such a lack of padding as to create "brick walls" of widgets. Applications which exhibit these qualities can be found at the Tcl/Tk archive site. Mega-widgets can lessen the occurrence of these visual works of art while maintaining simplicity and ease of use. For example, use of the [incr Widgets] ButtonBox mega-widget can stop the button "brick wall" effect.

Look-and-Feel

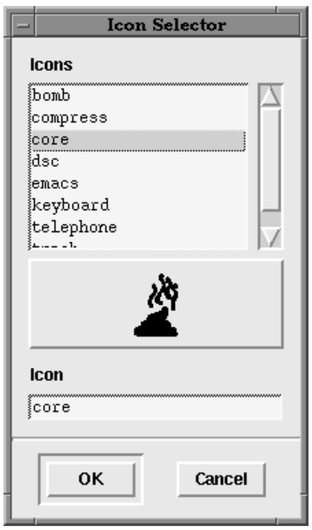
It is visually evident from the example that the look-and-feel of [incr Widgets] is Motif. Adherence to the style guide is close. Little effort has been spent attempting to make minor improvements. This is even more clear in some of the larger [incr Widget] dialog mega-widget classes such as the FileSelectionDialog. The Motif likeness is also evident in the initial selection of classes which comprise the mega-widget set, including most of the Motif favorites. It even extends beyond appearance to behavior and options.

The Motif look-and-feel was chosen because of its strength in the industry and customer requirements. The demand of the current [incr Widgets] customer base is a Motif appearance and behavior, regardless of the underlying implementation. Thus, very few liberties were taken in the visual style and behavioral aspects of [incr Widgets]. Instead, concepts such as extensible child sites and flexible component configuration option sets have been implemented which allow developer divergence from the Motif style on an as needed basis.

Extensibility

The extensibility of [incr Widgets] is based on a similar concept found in Motif called "child sites" which allow the basic functionality and visual appearance of an existing mega-widget to be augmented. The idea is simple, yet it yields a powerful mechanism by which mega-widgets become malleable and reusable. They allow for the possibility of unanticipated future requirements, making for a much less restrictive widget set.

Consider an application which requires a icon selection dialog, visually displaying the icon as the textual name is selected from the list. Also, suppose we would like to see this canvas on which the icon is presented lie between the listbox and entry widget. This could easily be implemented using the [incr Widgets] SelectionDialog mega-widget which maintains a child site, as depicted in Figure 10.




```
SelectionDialog .iconSel -disphelp no \
    -title "Icon Selector" \
    -itemslabel Icons \
    -selectionlabel Icon \
    -selectcommand SelectProc \
    -childsitepos center

set cs [.iconSel childSite]

canvas $cs.canvas -height 70 \
    -relief raised -borderwidth 2
pack $cs.canvas -fill x -expand yes

proc SelectProc {} {
    set c [.iconSel childSite].canvas
    $c delete all
    $c create bitmap \
     [expr [winfo width $c] / 2] \
     [expr [winfo height $c] / 2] \
     -bitmap @[.iconSel get].xbm
}

.iconSel insert items end bomb compress \
    core dsc emacs keyboard telephone \
    trash workstation

.iconSel activate
```

FIGURE 10 - Icon selector dialog

The advantages of child sites can be seen by examining the opposite situation. Had the SelectionDialog been designed minus a child site, the user would have been forced to create the icon selector from scratch or become aware of its internal packing and attempt to repack around the canvas. A take-it-or-leave-it design such as this would be limiting, sacrificing possibilities of reuse.

The means by which a child site may be filled is not a limiting factor in the [incr Widgets] set. Either composition or inheritance may be used. In the login screen example, both of these mechanisms were demonstrated. First composition was used. Later, as the Login mega-widget class was produced, the same child site was filled by means of [incr Tcl]'s inheritance feature. The implementation of child sites and the means by which they may be accessed in [incr Widgets] deserves closer inspection.

[incr Tk] has several base classes from which the [incr Widgets] class hierarchy is derived. As base classes they provide option management, standard methods, and a parent for components called the "hull" widget. The path to this widget is contained in a protected class variable named "itk_hull". Many mega-widgets within [incr Widgets] successively maintain this variable in the hierarchy. As a mega-widget is constructed, new components are built off the path stored in the "itk_hull" variable. The mega-widget may also construct a new hull and store its path in "itk_hull" for a future derived class to use. A 'childSite' method is provided for composition support.

Flexibility

Frequently, a mega-widget straight out of the box doesn't exactly fit the bill. Developers need to tweak the visual layout here and there to meet their application requirements. [incr Widgets] provides this capability with the viewpoint that flexibility yields reuse. Each mega-widget was designed to allow modification of the visual aspects of the components through a rich option suite. As with standard Tk widgets, options may be specified at construction time and subsequently there after with the 'configure' command.

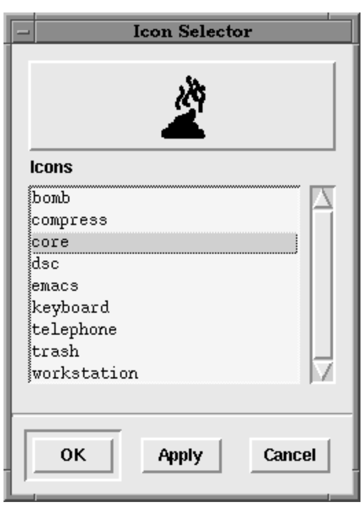
This is a very useful feature. For example, all the mega-widgets which support scrollbar attachment do so at the developer's discretion. One may choose to have each scrollbar independently displayed either statically, dynamically, or never. A dynamic scrollbar would appear as needed based on the number of elements in the widget and their ability to fit in the allotted space, whereas a static one is always displayed. Thus, scrollbars have built-in intelligence.

Similarly, ButtonBox usage is not limited to horizontal, distributed, equal width button management. Instead, buttons may be packed to a side, and the box itself may be oriented vertically as well as horizontally. Also, each button may be referred to by its label in commands which allow them to be hidden, shown, or made the default.

Flexibility is built into the larger scale mega-widgets as well. The SelectionBox class allows specification of labels, their position relative to their associated widget, and control over the display of the each element. The FileSelectionBox provides this same ability. Thus, the filter or selection labeled entries can be unmanaged as well as the file and directory lists.

Large scale flexibility presents significant advantages. Applications may be designed in a more interactive manner. Consider the icon selector dialog example again. Suppose an alternate presentation of the dialog was to be considered. One in which the icon canvas appears above the list, the selection entry widget is removed, and an apply button is added. This could be quickly examined by configuring the components as given in Figure 11.

The ability to reconfigure components also allows programs to be built which change appearance on the fly. An application which demands multiple flavors of a mega-widget with different looks can create one instance and change the options between uses. This can be much more efficient, since construction time is much more costly than the time required to configure and map the widget.




```
.iconSel configure -childsitepos n \
        -selectionon no -dispapply yes
```

FIGURE 11 - Alternate icon selector dialog

For example, consider an application which must confirm a user request prior to performing the operation. In addition, due to the serious nature of the operation the user must confirm positively twice. The following code segment creates the initial message dialog and configures the message to ask "Are you sure ?". The dialog is then mapped with the 'activate' command. If the user responds positively, then the message is changed to "Are you really sure ?" and redisplayed. Only with two affirmative replies does the script perform the operation. Figure 12 depicts the dialog presentations with different messages and the associated code.


```
MessageDialog .md -modality application \
    -disphelp no -title Confirmation \
    -message "Are you sure ?" -oklabel Yes \
    -cancellabel No -bitmap questhead

if {[.md activate]} {
    .md configure \
        -message "Are you really sure ?"

    if {[.md activate]} {
        # Perform operation
    }
}
```

FIGURE 12 - Confirmation dialog

This same instance of the message dialog can be reconfigured into a error dialog. All the options can be dynamically changed. It is possible to not only change the bitmap but its location as well. Furthermore, we can modify the text of the buttons and make the dialog nonmodal. We'll also change the orientation and position of the buttons to be vertical along the right hand side. Figure 13 shows the final product. It is important to note, that no new message dialog has been created, instead the existing one has been reconfigured.


```
.md configure -bitmap Error -bitmappos n \
    -message "Unable to access device" \
    -oklabel Retry -cancellabel Cancel \
    -modality none -buttonboxpos e \
    -orient vertical
.md activate
```

FIGURE 13 - Error dialog

Lessons Learned

One element which is essential to any successful development effort is the establishment of firm objectives. [incr Widgets] was short on neither aggressive goals nor talented developers willing to contribute in a team environment. Many lessons were learned during this effort as the team achieved a truly reusable, flexible, and extensible mega-widget set. The lessons centered on inheritance, configurability, testability, and reusability.

Inheritance proved to be a valuable tool during [incr Widgets] development. The impact of changes due to Tk 4.0 were significantly lessened. For example, as image support was added to Tk, a single option was added to the LabeledWidget class which was then inherited by derived classes in the hierarchy. Also, bugs fixed in base classes applied to all derived ones. This made for quick and easy maintenance. On a similar note, errors introduced in lower level classes had broader effects. Fortunately, this was rare and easily detected due to the magnified repercussions.

Maximum reconfigurability comes at the price of quickly multiplying options in an inheritance hierarchy. In an effort to avoid the usability problems associated with Motif's bulky resource set, [incr Widgets] imposed an 80/20 rule. If 80% of the user community could be viewed as having no interest in an option, it was excluded. Users could always use the built-in [incr Tk] 'component' command to configure an option.

The incorporation of a regression test suit was a definite plus. The [incr Widgets] test suite is a blatant rip-off of the work done by Ousterhout and May-Pumphrey for Tcl/Tk [6]. The test suite consistently exposed flaws which hand testing left hidden. Especially those bugs dealing with large scale component configuration. The test suite also doubles as a good visual demo.

Absolutely no reuse of any kind occurs until a widget set becomes well documented. This includes man pages, user's guides, and demos. Unless documented, reuse is a localized event at best. There is no such thing as self-documenting code. Instead, the demand is for self-documenting engineers.

Prospective

[incr Widgets] is an ongoing development effort. The mega-widgets presented in this paper represent those ready for release. Each has a man page, demo, and regression test script. Many other mega-widgets are under construction which have not reached release status. They include classes such as ToolBar, ComboBox, Table, Calendar, Gage, MenuBar, and MainWindow. Once complete, each new class will be incorporated into the [incr Widgets] distribution.

Public contributions to the [incr Widgets] mega-widget set are welcome and encouraged. Those mega-widgets which currently compose [incr Widgets] should be used as a model. Contributed mega-widgets should meet or exceed the objectives set forth in this paper such as extensible child sites, flexible component configurations, and style consistency. The coding and comment style must also be maintained. Man pages, demos, and test scripts are mandatory.

Conclusion

[incr Widgets] offers a strong object-oriented foundation which addresses the need for a flexible and extensible mega-widget set. Its usage replaces common widget combinations with higher level abstractions, simplifying code, reducing errors, increasing readability, adding productivity, and promoting a singular look-and-feel. The ability to extend [incr Widgets] enables developers to create new mega-widgets based on previous work.

In short, [incr Widgets] is a library of reusable mega-widgets that can be easily extended, allowing quicker development of large scale applications. It has been successfully used in several projects, including mission-critical telecommunication applications delivered to Japan, Great Britain, and Australia. As development continues, existing classes are being extended and new classes are being added. Development time has been drastically reduced. New dialogs can be created in hours. Whole applications in days. Reuse is a reality. New projects are benefitting from the work of others. [incr Widgets] is an [incr Tk] success story.

Acknowledgments

[incr Widgets] was produced by a dedicated team comprised of Mark Ulferts, Sue Yockey, Alfredo Jahn, and Bret Schuhmacher at DSC Communications Corp. Significant advice and counselling was administered by Mark Harrison, also employed by DSC. Michael J. McLennan, AT&T Bell Labs, allowed the team to beta test [incr Tk] and supported the effort through the infusion of innovative ideas.

References

[1] Michael J. McLennan, "[incr Tcl] - Object-Oriented Programming in Tcl", Proceedings of the Tcl/Tk Workshop 1993, Berkeley Ca. http://www.wn.com/biz/itcl

[2] Michael J. McLennan, "[incr Tk] Building Extensible Widgets with [incr Tcl]", Proceedings of the Tcl/Tk Workshop 1994, New Orleans La. http://www.wn.com/biz/itk

[3] Ioi K. Lam, Tix, http://www.cis.upenn.edu/~ioi/tix/tix.html

[4] Nat Pryce, itcl-widgets, http://www-dse.doc.ic.ac.uk:80/~np2/itcl_widgets/

[5] George Howlett, BLT-1.7, ghowlett@fast.net, 1994

[6] John Ousterhout, "Tcl and the Tk Toolkit", Addison-Wesley, 1994. http://playground.sun.com/~ouster/

Appendix
[incr Widgets] Tour

[incr Tk] provides the base classes for all the mega-widget classes of [incr Widgets]. The dialog classes are derived from itk::Toplevel, all other classes are based on itk::Widget. The [incr Tk] classes provide for component, option, and method definition and management. The [incr Widget] classes are the specialization of the [incr Tk] base classes, where each level refines and augments the methods and options of the base classes. Each [incr Widgets] class will be briefly discussed and a short example presented. Figure 14 depicts the class hierarchy.

LabeledWidget

The LabeledWidget is the most primitive mega-widget in the set, providing label support in the other classes. The class contains a label, a margin, and a child site which can be filled with other widgets. The options provide the ability to position the label around the child site, modify the font, adjust the margin distance, and enable/disable label display.

The following example creates a LabeledWidget with a canvas widget in the child site. The label is set to "Canvas" and initially located south of the child site. Next, the label is moved around the child site and margin set to various distances.

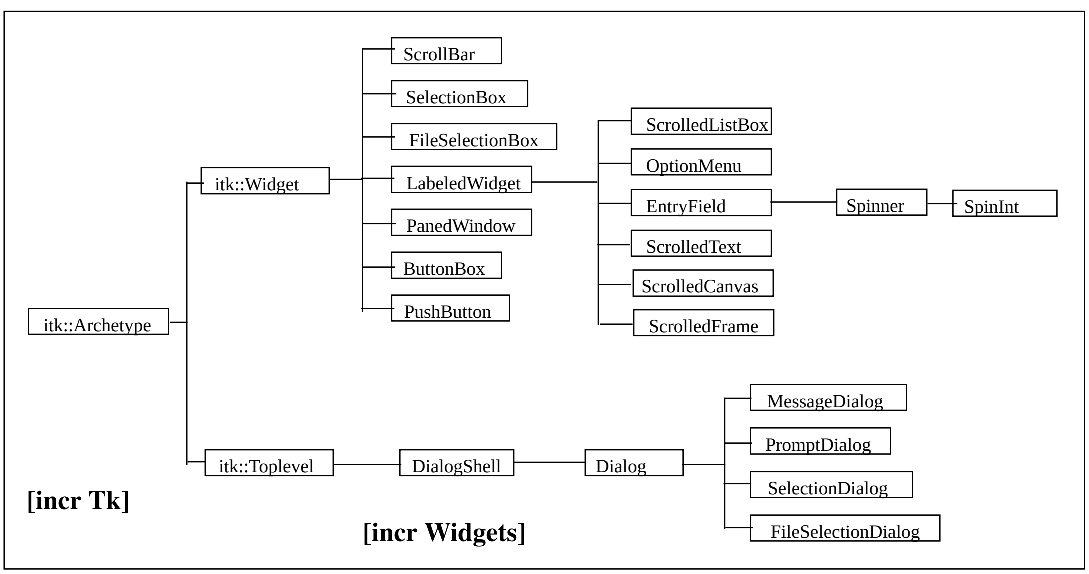
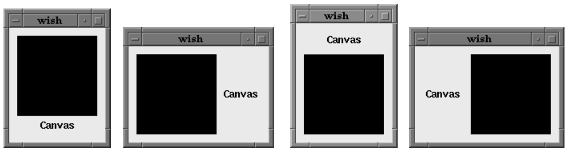


FIGURE 14 - [incr Widgets] class hierarchy




```
LabeledWidget .lw -labeltext “Canvas” \
   -labelon yes -labelpos s
set childsite [.lw childSite]
canvas $childsite.c -relief raised \
   -width 100 -height 100 -background black

pack $childsite.c
pack .lw -fill both -expand yes \
   -padx 10 -pady 10

.lw configure -labelpos w -labelmargin 10
.lw configure -labelpos e -labelmargin 5
.lw configure -labelpos n -labelmargin 7
```

FIGURE 15 - LabeledWidget

EntryField

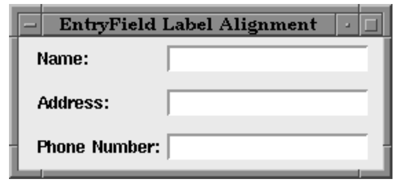
The EntryField class associates a label with an entry widget, providing text entry, length, validation, and editing enhancements. Since the class is based on the LabeledWidget class, all the options and methods for LabeledWidgets are supported in EntryFields. Also, most of the methods for the standard Tk entry widget are provided such as insert, delete, get, and scan.

PushButton

The PushButton class offers the standard Tk button widget with the ability to display as a default button with a recessed ring. The primary use for the PushButton is as a child of the ButtonBox class. In addition to furnishing the standard methods and options for Tk button, the PushButton provides options for enabling/disabling the display of the default ring and geometry requirements.


```
PushButton .pb -text PushButton \
    -defaultring yes -defaultringon yes
pack .pb -padx 12 -pady 12
```

FIGURE 17 - PushButton




```
EntryField .name -type alphabetic \
    -labelon yes -labeltext Name:

EntryField .address -type alphanumeric \
    -labelon yes -labeltext Address:

EntryField .phone -type numeric \
    -labelon yes -labeltext “Phone Number:”

LabeledWidget::alignLabels \
    .name .address .phone

foreach wid [list .name .address .phone] {
    pack $wid -pady 5 -padx 10 \
        -fill x -expand yes
}
```

FIGURE 16 - EntryField

OptionMenu

The OptionMenu class allows selection of one item from a set of items. Only the selected item is displayed, until the user selects the option menu button and a popup menu appears with all the choices available for selection. Once a new item is chosen, the currently selected item is replaced and the popup is removed from the display. Commands exist for manipulating the menu list contents as well. These include the ability to insert, delete, select, disable, enable, and sort items.


```
OptionMenu .om -labelon true \
    -labeltext “Operating Systems:” \
    -items {SunOS HP/UX AIX OS/2 Windows} \
    -command SelectProc
pack .om -padx 10 -pady 10

.om insert end Linux VMS
.om disable item DOS
.om delete 1 2
.om sort ascending
.om select item Linux
.om configure -cyclicon true
```

FIGURE 18 - OptionMenu

Spinner

Spinners constitute a set of widgets which provide EntryField functionality combined with increment and decrement arrow buttons which may be oriented in a vertical, top and bottom, fashion or in a horizontal, side by side, manner. A value may be entered into the entry area explicitly or the buttons may be pressed which cycle up and down through the choices. This latter behavior is one of spinning.

The following code segment creates a month spinner based on the Spinner class. The months are stored in a list from which the spinMonth procedure cycles. The Spinner increment and decrement options invoke this procedure with a direction argument which is 1 or -1. The Spinner disables input by making the blockInput procedure the validation procedure which always returns invalid.

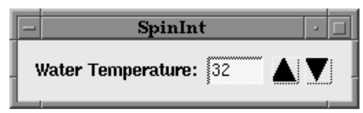

```
set months {January February March April \
           May June July August September \
            October November December}

proc blockInput {char} {return 0}

proc spinMonth {direction} {
    global months
    set index \
      [expr [lsearch $months [.sm get]] + \
             $direction]

    if {$index < 0} {set index 11}
    if {$index > 11} {set index 0}

    .sm delete 0 end
    .sm insert 0 [lindex $months $index]
}

Spinner .sm -labelon yes \
    -labeltext “Month : “ -width 10 \
    -fixed yes -type other \
    -validate blockInput \
    -decrement {spinMonth -1} \
    -increment {spinMonth 1}
.sm insert 0 January
pack .sm -padx 10 -pady 10
```

FIGURE 19 - Spinner

SpinInt

The most common data type for which spinning behavior is useful is that of integers. The SpinInt class offers this capability by specializing the Spinner class. Additional options are provided which allow specification of a step and range values which vary and limit the cycling. A boolean wrap option also exists which stipulates the action to be taken upon reaching the minimum and maximum values.

The following example creates a water temperature integer spinner widget which is labeled appropriately. The widget options limit the range of values to between freezing and boiling, specifies a step value of two, enables wrapping, and orients the buttons in a side by side horizontal fashion.




```
SpinInt .temp -labelon yes -labelpos w \
    -labeltext “Water Temperature:” \
    -fixed yes -width 5 -range {32 212} \
    -step 2 -wrap yes -orient horizontal
pack .temp -padx 10 -pady 10
```

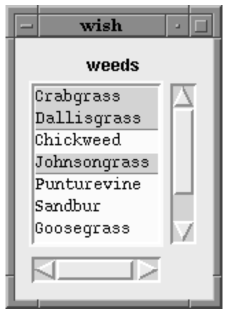
FIGURE 20 - SpinInt

ScrollBar

The ScrollBar class is the standard Tk scrollbar with a couple of variances. The foreground option has been removed and the background option modified to set the elevator and arrows to the specified color and the trough to 80% of this value. Those [incr Widgets] classes which need scrollbars use the ScrollBar class.

ScrolledListBox

The ScrolledListBox extends the standard Tk listbox widget with predefined vertical and horizontal scrollbars and an associated label. The set of options available has also been amended to include options which allow specification of the list items and designate procedures for single and double click selections. All the usual methods exist, plus new ones for sorting the list contents and a short cut version to acquire the current selection.

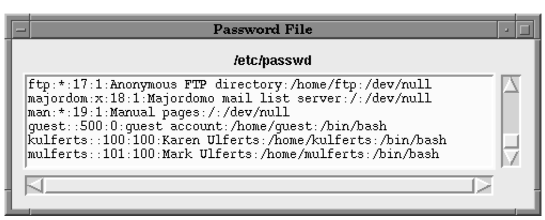



```
ScrolledListBox .slb -vscrollmode static \
  -hscrollmode dynamic -selection SelProc \
  -items {Crabgrass Dallisgrass Nutsedge} \
  -scrollmargin 5 -labelon yes \
  -labelpos n -labeltext "Weeds"
pack .slb -padx 10 -pady 10

.slb insert 2 Sandbur Goosegrass
.slb insert end Chickweed Johnsongrass
```

FIGURE 21 - ScrolledListBox

ScrolledText

The ScrolledText widget provides all the functionality of the standard Tk text widget along with scrollbar and label control. The set of methods has been extended to include import and export file capabilities.

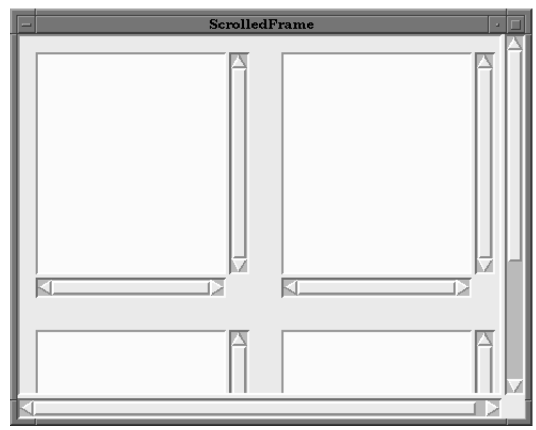



```
ScrolledText .st -labelon yes -labelpos n \
        -labeltext "/etc/passwd" \
        -vscrollmode static \
        -hscrollmode static
pack .st -padx 10 -pady 10 -fill both \
         -expand yes

.st import /etc/passwd
```

FIGURE 22 - ScrolledText

ScrolledFrame

The ScrolledFrame combines the functionality of scrolling with that of a typical frame widget to implement a clipable viewing area whose visible region may be modified with the scrollbars. This enables the construction of visually larger areas than which could normally be displayed, containing a heterogenous mix of widgets. Once created, the ScrolledFrame child site can be accessed and filled with widgets.




```
ScrolledFrame .sf -vscrollmode static \
                  -hscrollmode static \
                  -width 475 -height 360
set childsite [.sf childSite]

pack [frame $childsite.topframe]
pack [frame $childsite.botframe]

ScrolledListBox $childsite.topframe.slb1
pack $childsite.topframe.slb1 -side left

ScrolledListBox $childsite.topframe.slb2
pack $childsite.topframe.slb2 -side left

ScrolledListBox $childsite.botframe.slb3
pack $childsite.botframe.slb3 -side left
ScrolledListBox $childsite.botframe.slb3
pack $childsite.botframe.slb3 -side left

pack .sf -expand yes -fill both
```

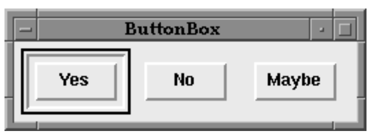
FIGURE 23 - ScrolledFrame

ScrolledCanvas

The ScrolledCanvas applies scrollbars and display options to a standard Tk canvas widget. All the standard canvas commands and options have been maintained. A new option, autoresize, has been added which allows the user to engage automatic resizing of the scroll region to be the bounding box covering all the items. The region is adjusted continuously as items are created and destroyed via the canvas commands, effecting the display of the scrollbars.

ButtonBox

The ButtonBox performs geometry management for PushButton instances. Public commands exist which enable the user to add new PushButtons, define the default, and control their display. Options enable the user to specify the packing mode of the PushButtons and establish the orientation. This class is used to manage the buttons for all dialogs in the [incr Widgets] mega-widget set.




```
ButtonBox .bb -padx 10 -pady 10

.bb add -text Yes
.bb add -text No
.bb add -text Maybe
.bb default Yes

pack .bb -expand yes -fill both
```

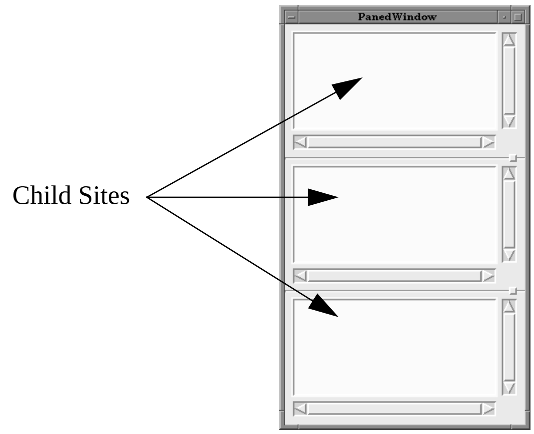
FIGURE 24 - ButtonBox

PanedWindow

The PanedWindow class is composed of panes, separators, and sashes for adjustment of the separators. Each pane is a child site and the class provides a command which returns the paths for the sites. The user may fill them with further widget combinations.

To illustrate child site access, the following code segment creates a PanedWindow of three panes. Next the ‘childSites’ command is used to retrieve a list of the pane path names. The ‘foreach’ loop then cycles though the list and creates a ScrolledListBox in each pane.

The PanedWindow offers significant control over its presentation through a large set of options. The option set allows specification of the distance between a pane and its contents, the minimum size a pane’s contents may reach, the orientation of separators, the number of panes, the thickness of the separators, as well as the dimensions, position, and cursor associated with the sash.

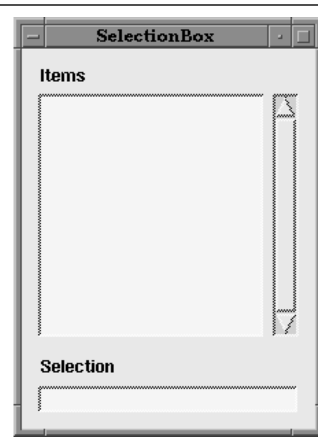



```
PanedWindow .pw -width 300 \
    -height 500 -panes 3
pack .pw -fill both -expand yes

foreach pane [.pw childSites] {
    ScrolledListBox $pane.slb \
        -vscrollmode static \
        -hscrollmode static
    pack $pane.slb -fill both -expand yes
}
```

FIGURE 25 - PanedWindow

SelectionBox

The SelectionBox class combines a scrolled list of items, an editable entry field for the selected item, and labels for the list and entry field, allowing the user to select or enter one item from a list of alternatives. The SelectionBox also provides a child site and an option to control its position.

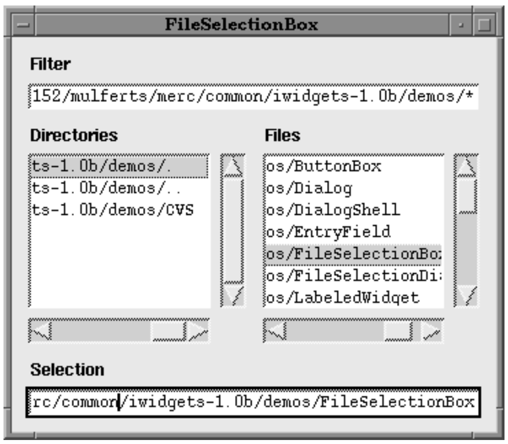



```
SelectionBox .sb
pack .sb -padx 10 -pady 10
```

FIGURE 26 - SelectionBox

FileSelectionBox

The FileSelectionBox presents a file selector similar to that found in the Motif widget set. It consists of a file and directory list as well as a filter and selection entry widget. A child site also exists which may be positioned at several locations via an option. An extensive option set is provided which enables specification of initial directory, search commands, filter mask, no match string, and margins.




```
FileSelectionBox .fsb
pack .fsb -padx 10 -pady 10
```

FIGURE 27 - FileSelectionBox

DialogShell

The DialogShell class provides base class support for top level [incr Widgets] modal dialogs. This includes dialog mapping, button management, separator control, and a child site. The ‘activate’ command maps the dialog and waits based on the modality. Non-modal dialogs return control immediately, whereas system and application modal dialogs wait until the ‘deactivate’ command is invoked. The ‘deactivate’ command accepts an optional argument which becomes the return value of the ‘activate’ command. This provides dialog deactivation status notification.

Dialog

The Dialog class is a specialized version of the DialogShell with four predefined buttons, “OK”, “Cancel”, “Apply”, and “Help”. Options control their display and associated commands. By default the Dialog class provides automatic deactivation and status return following selection of either the “OK” or “Cancel” button. The status indicator is 1 for “OK” and 0 for “Cancel”. Automatic deactivation may be disabled, enabling more user control over deactivation and status notification. In this case, the user must invoke the deactivate method explicitly and is free to pass a status return value as needed.

MessageDialog

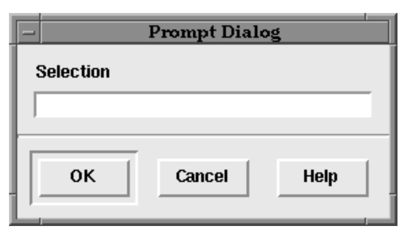
The MessageDialog class provides a bitmap and message text within a dialog context. Options control the position of the bitmap and message. All the standard dialog control options are also available.


```
MessageDialog .cr -title Copyright \
    -bitmap @dsc.xbm -imagepos n \
    -dispapply no -dispcancel no \
    -disphelp no -text "Copyright 1995\
      DSC Communications Corporation\n\
      All rights reserved"
.cr activate
update
after 10000 ".cr deactivate"
```

FIGURE 28 - MessageDialog

PromptDialog

Based on the Dialog class, the PromptDialog provides a Motif style prompt dialog.

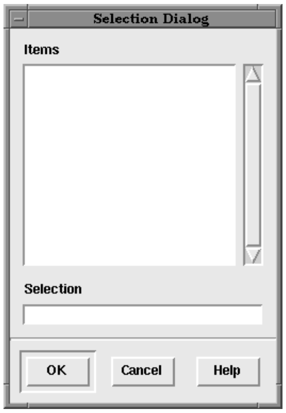



```
PromptDialog .pd
.pd activate
```

FIGURE 29 - PromptDialog

SelectionDialog

The SelectionDialog class provides a dialog based SelectionBox.

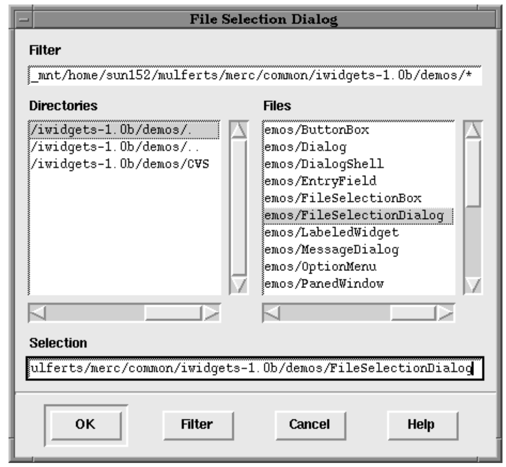



```
SelectionDialog .sd
.sd activate
```

FIGURE 30 - SelectionDialog

FileSelectionDialog

The FileSelectionDialog is a dialog based FileSelection-Box.




```
FileSelectionDialog .fsd
.fsd activate
```

FIGURE 31 - FileSelectionDialog