# Fileselector - A Widget Cookbook

*Jacco van Ossenbruggen*

Assumes (unreleased) hush 2.1

## Abstract

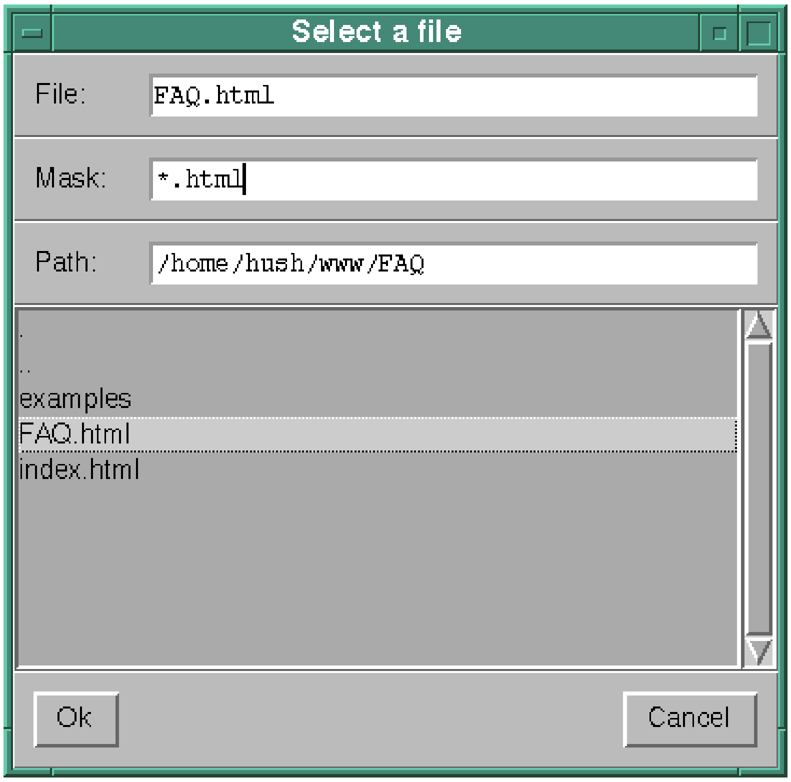
The hush library offers excellent support for extending Tcl/Tk with new, user-defined widgets. This article will explain how to define a new, composite widget in hush, and how to define a proper interface in order to be able to easily access the widget from (compiled) C++ code as well as (interpreted) Tcl code.

## Introduction

We will assume the reader has some experience in using plain Tcl/Tk as well as C++. All code examples can be found in the directory `/home/hush/tutorial/fileselector/src1`. Code examples are written in C++, unless stated otherwise. Individual shell commands will be preceded by a prompt indicating the name of the shell used, e.g:

```
sh> pwd                       # Bourne shell command line
wish> button .b               # wish command (real wish prompt is usually '% ')
hush> hypertext .h            # hush command
fs-hush> fileselector .f      # command to an interpreter called fs-hush
```

As a running example, we will develop a fileselector widget as shown in figure 1.




The widget is a composite widget, build of several standard Tk widgets. In the next section we will briefly sketch the look and feel of the fileselector and the Tk widgets which constitute our composite widget.

## First things first: Prototype!

The fileselector's GUI can be defined in Tcl/Tk without using the hush library, and the reader is encouraged to spend some time playing with the standard Tcl/Tk interpreter: `/usr/local/bin/wish`, and try to build a GUI like the one in figure 1. An example Tcl/Tk script building the GUI of our fileselector is listed in Appendix A.

In general, it is good practice to prototype a GUI in an interpreted language like Tcl. Buildings GUIs is often a process of trial and error and the huge overhead associated with recompiling and relinking each time you make minor modifications becomes very annoying.

Fortunately, Tcl/Tk interpreters such as wish can be used in an "interactive mode", allowing the user to start the interpreter with a small, empty window, and add new widgets by typing commands at the the wish prompt. The interactive mode often proves to be a better environment to try things out than C++!

A sequence of Tcl/Tk commands can be collected in a file (usually with a `.tcl` suffix) and executed by typing:

```
sh> wish file.tcl
```

The file itself can be made an executable wish script by including the following as the first line of the file (do not forget to chmod your file):

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

A third possibility is to load the source file from another Tcl/Tk script, or directly from the wish prompt:

```
wish> source file.tcl
```

Note that every hush application is a fully-fledged Tcl/Tk interpreter, so you can use your own hush applications to prototype interfaces!

# The Graphical Interface

The example widget is composed of other, standard Tk widgets. We will shortly discuss the widgets used. More information on the individual widgets can be obtained from section *n* of the Tcl/Tk manual.

**Toplevel**

The fileselector is displayed in a new window, which is achieved by packing all components in a toplevel(n) widget.

**Entry**

The three white boxes at the top of the figure are entry(n) boxes. The user can enter the filename, mask and directory path directly into these three widgets.

**Label**

The meaning of the several entries is clear by the use of the three label(n) widgets at the left of the entries. Labels can show arbitrary text strings, in this case "File:", "Mask:" and "Path:".

**Frame**

Each entry box and its corresponding label are grouped together in a frame(n) widget. Framing is a common structuring mechanism in Tk, primary used to enhance the way widgets are packed to the screen. See Appendix A for more information on packing.

Frames are often not explicitly visible, but in this case they are: note the thin horizontal line between each entry box.

**Listbox**

The largest and most important widget is the listbox(n), displaying the contents of the current directory. Selecting (by double clicking) a file from the listbox should probably update the file entry as well, and selecting a directory will update both the path entry and the listbox itself.

**Scrollbar**

A scrollbar(n) widget is supplied in order to deal with large directories. The scrollbar and the listbox are also packed in a frame.

**Button**

Finally, the bottom of the fileselector shows a "Cancel" and "Ok" button(n), that are packed in a frame as well.

### User Interaction: Binding Events

Now we have defined the component widgets constituting our fileselector, it is time to define the behavior of all components. Some of them (e.g. labels, frames, toplevel), are not able to react on user interaction, but most of them are. Every user interaction is modeled by an *event*. An event can represent hitting a key, pressing or releasing a mouse button, moving the mouse, etc. The relation between an event and a procedural action is called a *binding*. See the Tk bind(n) command for more information.

Most events are bound to pieces of program code which cause changes to the associated widget itself, the other widgets or, more general, the state of the application.

For instance, hitting the return key after typing a new directory path in the path entry causes a change in the current working directory of the application, an update of the filenames in the listbox, and the deletion of any text in the filename entry.

There are several events that cause the fileselector's window to disappear because the user wants to end the dialog. These events are: pressing the OK or Cancel button, hitting the return key in the filename entry and double clicking on a filename in the listbox.

Double clicking on a directory name in the listbox causes a directory change and an update on the path entry and listbox, hitting return on the mask entry causes an update on the listbox, etc.

In the prototyping phase, it may be convenient to define this behavior in the Tcl language as well. However, in general it is more practical (especially if more complex behavior has to be defined) and efficient to be able to use the expressive power of a compiled language like C++. The hush library gives excellent support to bind C++ code --- type safe --- to events. See Appendix C for more details.

## Using Tcl/Tk in C++

To be able to use the facilities of Tcl/Tk in C++, the hush API provides two class libraries:

- The **hush** library contains some classes representing some general concepts, which are partially inspired by the InterViews toolkit. The kit(4) class encapsulates the underlying Tcl/Tk toolkit. More information about the individual hush classes can be found in section 4 of the hush manual.
- The **widgets** library contains class definitions for all Tk widgets. More information about the individual widget classes can be found in section 5 of the hush manual. The widgets library allows a C++ programmer to build a Tk-based GUI directly in C++. Most of hush' widgets are small and cheap wrapper classes, delegating their functionality directly to the underlying Tcl/Tk interpreter by using the hush kit(4). You might want to trace which Tcl/Tk commands are executed by your

application for debugging purposes. This can be achieved by calling the `kit::trace()` member.

## The fileselector class

Since we want to be able to use our fileselector in C++ as well, we will define a C++ class with the same name. The class interface of our fileselector is fully discussed in Appendix B.

The fileselector's class interface consist of more than GUI related member functions. The partial class definition below lists the interface representing the other functionality. It contains functions to get and set the filename mask and the pathname of the current directory, a rescan operation and a function returning the selected filename (the filename is what this is all about, remember!). Note that all these functions are declared `virtual` so they can be redefined by other fileselector classes, derived from this one. However, since many of the members functions are called by other member functions (e.g. the `mask(const char*)` functions makes a call to `rescan()` to reflect a change of the filename mask), the new implementations will have to display comparable behaviour.

```
interface fileselector: public toplevel {
    public:
    // Typical hush and GUI stuff deleted:
    ...
    // Typical fileselector stuff:
    virtual void mask(const char *f);   // set filename mask (default "*")
    virtual const char* mask() const;   // get    ,,      ,,
    virtual int dirpath(const char* path); // set current dir path
    virtual const char* dirpath() const ;  // get current dir
    virtual void rescan();              // list and display current dir
    virtual const char* get();          // get (and wait for) filename
    };
```

### Using C++ classes to handle new Tcl commands

To be able to use the fileselector in the same way as the build-in widgets, we want to have a new Tcl command with the same name. If you have a handler(4) object which can handle a new command, the command can be introduced to the Tcl/Tk toolkit by use of the kit::bind(cmdname, handler) function. Since our fileselector is (indirectly) derived from handler and designed to handle the fileselector command, this is can be considered trivial. See fs-hush.cc for an example.

Note on implementation: Installing the command requires the creation of an instance of the fileselector class to handle the Tcl command with the same name. However, the creation of this instance should not result in the creation of a real fileselector widget on your screen! To achieve this, a special "dummy" fileselector widget is created. In hush, it is common practice (it should be!) to define default constructors which create the dummy widgets dealing with the associated command. See Appendix B and C for more details.

### Using C++ classes to override old Tcl commands: rename()

Suppose we have created a fileselector widget named `.fs`:

```
fs-hush> fileselector .fs
```

What the command does (whether it is implemented in C++ or Tcl) is creating the widget hierarchy, which has a toplevel(n) widget with the name `.fs` as its root. Consequently, a plethora of Tcl commands, with names equal to the pathnames of all the widget's windows, are introduced (e.g .fs, .fs.boxframe, etc). While there might be practical reasons for allowing this pollution of the global command name space, the fact that the .fs command refers the toplevel window is highly inconvenient. We want to be able to do things such as:

```
fs-hush> fileselector .fs
fs-hush> set filename [ .fs get ]
```

However, since ".fs" refers to the toplevel window, we will get an error message since the get command is not defined for toplevels. What we want to do is rename the toplevel and bind the fileselector object to the name ".fs". To to this, we use the following C++ code:

```
// somewhere in a member of fileselector:
...
toplevel* top = new toplevel(some_parameters);
...
top->rename();      /// rename widget command name (must be unique)
tk->bind(this->name(), this); /// bind handler to widget command name
```

Since we do not really care what the new name of the toplevel will be (provided it is unique), we do not supply an argument to the rename function (hush will generate a new unique name by postfixing the old one with '-cmd'). So the top->rename() call will rename the toplevel widget command name from .fs to .fs-cmd. This allows us to use .fs as the command name for the fileselector, which we bind by the tk->bind(this->name(), this) call.

*Warning*: Note the difference between the Tk path name of a widget and the associated Tcl command name. The rename function will not change the Tk path name, it will only change only the Tcl command name. E.g:

```
wish> button .foo;
wish> rename .foo .bar
wish> pack .foo         # OK
wish> pack .bar         # Error: pathname is .foo
wish>
wish> .foo configure -text "Hello"      # Error: command name is .bar
wish> .bar configure -text "Hello"      # OK
```

# Acknowledgements

Martijn van Welie and Bastiaan Schönhage designed and implemented the initial version of the fileselector code. Jelle Paul Alten made various contributions to the refinement of the code (especially he designed a variant using jumptables to implement event dispatching). Sebastiaan A. Megens and Gerard Sentveld critically reviewed earlier versions of this paper.

# Appendix A: Tcl code examples

The GUI of the fileselector can be defined by a Tcl script. Note that the code needed to create all widgets more or less balances the code needed to pack them properly into the window. The different

`-fill` and `-expand` options given to the pack command reflect differences in the space occupied by the widgets and differences in resizing when the window itself is resized by the user. For instance, the entry widgets should only resize in the horizontal dimension, the scrollbars only in the vertical dimension, and the listbox in both dimension. Other widgets, like the labels and the buttons or not supposed to scale at all.

*Tricky detail:* Note the use of the -before option in the pack configure command. It is used to make sure that the frame containing the Ok and Cancel button (allthough it is packed in the bottom) is inserted in the list of top's children *before* the frame containing the scrollbar and listbox (called boxframe). This order is important if the filechooser's window is resized by the user and the window becomes too small to contain all widgets. If this happens, the available space will be given to the first widgets in the list, and the size of the widget (s) at the end of the list will be reduced and eventually be removed from the screen! Because we do not want our buttons to disappear, we will place the boxframe at the end of the list. You can see the effect when scaling down the filechooser: The two widgets in the boxframe (the listbox and the scrollbar) will get smaller and eventually disappear. The same yield for the listbox and the scrollbar on an individual bases. If the scrollbar is added as as the last widget to the boxframe, the small scrollbar is removed from the screen when the window gets too small. In this case, we can solve this simply by packing the scrollbar before the listbox. More information on packing can be found in the packer tutorial.

# Appendix B: C++ class definition

The definition of the fileselector class in C++.

# Appendix C: C++ class implementation

The implementation of the fileselector class in C++.

# Appendix D: Small C++ example using fileselector class

Example C++ code.