Customization and Flexibility in the exmh Mail User Interface

Brent Welch
Sun Microsystems Laboratories
Mountain View, CA

Customization and Flexibility in the exmh Mail User Interface

Brent Welch
brent.welch@eng.sun.com
Sun Microsystems Laboratories
2550 Garcia Ave. M/S UMTV29-232
Mountain View, CA 94043

Abstract

The exmh mail user interface is designed to be flexible and customizable. The goal in the design is to provide a platform that can be tuned to suit individual preferences, as well as extended to provide custom functionality. There are several mechanisms for customization: install-time settings define runtime support requirements; a Preferences user interface exposes various "knobs" and "dials"; the X resource database controls fonts, colors, and other widget attributes; a personal library of Tcl procedures supports new functionality; the buttons and menus in the interface are defined by X resource specifications to allow customization and access to personal Tcl extensions; hook points in the implementation support callbacks to user extensions at key moments in mail handling; A binding user interface defines editing and accelerator keystrokes. This paper describes these mechanisms, and provides some experiences in the use and development of exmh.

1 Introduction

Exmh is a user interface to MH mail. MH was developed at RAND has been freely available for several years. It is a collection of UNIX programs that manage your mail. MH uses the file system for its state: a mail folder is a directory, each message is a file in a directory, and a few adjunct files maintain profile and context information such as the current folder. Instead of running one monolithic mail program, you can use the MH programs individually from the UNIX command line. Or, you can assemble them back into a monolithic mail reader that is tuned for your needs. The building-block nature of the MH programs make them well suited for composition with a Tcl script and a Tk graphical interface.

Exmh started as a short script I got from Ed Oskiewicz. It provided a starting point as I had not used MH before that time. I debugged, enhanced, and used exmh myself for several months before I decided to give it out to the rest of the Tcl community. I realized that exmh needed to be flexible. After all, the reason I was using it was that I wanted control over my mail reading environment. I also did not want to implement all the features that would inevitably be requested once other folks began using it. There were three means of customization in the initial version: the installer for system dependencies, a preferences package, and support for a library of personal Tcl code. The preference package exposed several "knobs" and "dials" that I thought users might want to change. The Tcl library was set up so users could copy parts of the implementation and fix or enhance them without changing the main application. In support of that goal, I broke the implementation up into several modules instead of having one large script.

The remainder of this paper describes how these customization features evolved, and the new customization features that were added to increase the flexibility of the system. The paper concludes with a few remarks about the experience of developing and maintaining exmh. As of this writing, over 800 different sites pick up each release of exmh, and almost 700 users have taken the time to register as an exmh user. I estimate there are a few thousand *exmh* users in total.

2 Installation and System Dependencies

Exmh is built on top of several packages, including MH [Peek95], Tcl/Tk[Ousterhout94][Welch95], Metamail (for MIME support), PGP (for public key encryption and digital signatures) [Garfinkle95], Glimpse (for full text indexing) [Glimpse94], and the facesaver database. To use these packages, *exmh* must know where they live in the file system. Instead of defining several environment variables that record this information, the locations are patched into *exmh* during installation. This is done with a simple installation program that is written as a Tcl/Tk script. After unpacking the *exmh* tar file, you run its installer like this:

```
% wish -f exmh.install
```

The implementation of the installer is table-driven. The `exmh.install` script consists of a series of directives like this:

```
install_var exmh(maintainer) \
    brent.welch@sun.com \
    {Target of error mailings}
install_dirVar mh_path /usr/local/bin
  {MH binary directory}
```

These are Tcl commands that take three arguments. The first argument is the name of a Tcl variable or array element, the second is its default value, and the third is a short text label for the installer user interface. These commands result in Tcl variable definitions in the final *exmh* script. The `install_var` command is the basic operation, and there are several variations that imply types for the variable values. These include `install_dirVar`, `install_fileVar`, and `install_progVar` that define directories, files, and executable programs, respectively. The install script does some simple checks to ensure that the values are ok.

The files to be installed are defined with two directives. `install_dir` specifies a directory, and `install_glob` defines a file name pattern of sources from the distribution that must be copied to the directory. (These two commands could be combined by adding the pattern to `install_dir`.) For example:

```
install_dir man /usr/man/manl \
    {Install man directory}
install_glob man exmh*.l
```

There are a few more specialized commands used to define help text and set up the testing environment used before the package is installed. Their details are not that important here. The main idea is to capture all the steps of installing a Tcl/Tk program, and to expose any dependencies on other packages to the system administrator.

The install package uses the directives to build a simple user interface that lists each value that must be set. There are buttons labeled `Patch`, `Test`, `Verify`, `Install`, `Config`, and `Quit`. The `Patch` operation uses *sed* to insert the Tcl variable definitions into the main *exmh* script. It is at this time that the semantic checks on the settings are made. Any errors are displayed to the user. This step also fixes the pathname for *wish* that appears in the first line of the script. Even if the install script gets lost, the system dependencies appear clearly marked right at the beginning of the *exmh* script.

The `Test` button runs *exmh* without installing it. This uses the script library as unpacked from the tar file. The `Verify` operation lists the commands that will be issued when the user clicks `Install`.

The `Config` button is used to select an existing set of definitions. A side effect of `Patch` is to save the settings into a `.exmhinstall` file. The installer looks in the current and peer directories for this file. This makes it easy to install new versions if you have installed *exmh* before.

You can pick up the `exmh.install` script and the supporting `install.tcl` Tcl procedures from:

```
ftp://parcftp.xerox.com/pub/exmh/
```

3 The Preferences Package

User preference settings are common in any large application. There are always design decisions that can go either way, and some users prefer it one way while others do not. For example, should "next message" skip over messages that are marked for delete, or should they be displayed? What command should be used to print a message? What editor do you want to use?

The *exmh* preference packages has three types of preference items: booleans, choices, and general strings. In the user interface, a checkbutton represents a boolean, a set of radiobuttons represent a choice, and an entry represents a general string. This is a fairly obvious approach. Sets of preference items are registered together under a heading such as "Background Processing" or "Folder Display", and the preference interface is composed of one main menu and then a toplevel window for each preference set.

A few issues arose during the design of the preference package:

- Definition of preference items.
- Management of a large number of settings.
- Representation for the settings for long term storage.

3.1 Definition of Preference Items

A set of preference items is registered with the `Pref_Add` command. It takes as arguments the name of the set, some general information, and a list of preference items. Each item is described by a Tcl variable, an X resource name, which is discussed later, an identifying label, and some additional help text. The user interface is generated automatically from this specification.

A call to `Pref_Add` looks like this:

```
Pref_Add title text listOfItems
```

Each item is a list of 5 elements:

```
var xres default label helpText
```

Here is an example:

```
Pref_Add "Background Processing" \
"A second process is used to perform various background tasks for Exmh. These options control its behavior." {
{exmh(bgAsync) bgAsync ON
    {Separate background process}
    "This setting determines whether or not the background processing is done in a separate process or not. A separate process causes less interference with the user interface, but might take up more machine resources."}
{exmh(background) bgAction
    {CHOICE off count msgchk flist inc}
    {Background processing}
```

```
    "exmh can periodically do some things
   for you:
    count - count new messages sitting in
   your spool file.
    msgchk - run the MH msgchk program.
    flist - check for new mail in all
   folders.
    inc - just like clicking the Inc but-
   ton yourself.
    off - do nothing in the background."}
 {exmh(bgPeriod) bgPeriod 10
    {Period (minutes)}
    "How often to do background task"}
 }
```

3.2 Managing Many Preference Items

The table driven approach to preference items makes it easy to add more. The original implementation did not divide the items into sets. As *exmh* evolved, however, the number of preference items grew large enough to become unwieldy. Among other things, a single window that displayed all the items was too large for some screens! A relatively minor change to the package was required to support a two-level scheme. Each module of the implementation already registered a set of preference items with a single call to `Pref_Add`. Its interface was changed to accept a name for the set and some overall information, and the layout of the display was changed to a menu with separate windows for each set.

The main drawback with the current approach is that it is implementation-based. The sets correspond to *exmh* modules that register preference settings. There are about 15 different preference sets, and it can be hard to find a particular setting. This is not inherent in the preferences package, however. It would be easy to add a search mechanism and an alternate display that showed all the settings in a compact manner. You could also reorganize the calls to `Pref_Add` to reflect the way the user thinks as opposed to the way *exmh* is built.

3.3 Storing Preference Data

The first version of the preference package saved the settings as a set of Tcl commands that initialized variables. The only trick is the proper formatting of the commands. Assuming the name of the preference variable is in `$varName`, the following commands work. The `list` command is essential to allow for arbitrary values:

```
upvar #0 $varName value
puts $prefFile \
    [list set $varName $value]
```

However, as described in the next section, *exmh* is also affected by a set of per-user X resource specifications for fonts, colors and other widget attributes. It seemed odd to users to have a `~/.exmhpref` for preference settings and a `~/.exmh-defaults` for resources. I was reluctant to drop the Tcl commands approach, but I finally gave in to user demand.

The default settings for *exmh* are stored in `app-defaults`, which is kept in the script library. Settings that are particular to a color display are in `app-defaults-color`, and settings for black and white displays are in `app-defaults-mono`. Personal settings mirror this arrangement. The `~/.exmh-defaults` file has preference settings and other generic resources. `~/.exmh-defaults-color` and `~/.exmh-defaults-mono` are used for color and monochrome display settings, respectively. Recently an additional set of files were added, `local-app-defaults`, `local-app-defaults-color`, and `local-app-defaults-mono`. This is for site-specific settings so that administrators can maintain a standard set of preference settings without modifying `app-defaults`. The three sets of files correspond to "factory settings", "site settings", and "personal settings". Tk associates a priority with resources, and the personal settings have the highest priority while the factory settings have the lowest priority.

When writing out the preference settings as resource values there is no need to worry about quoting. However, there is no support for resource values that contain leading blanks. The X resource database file format does not support it, and I did not invent any special coding for it.

The following commands generate a resource file specification. The leading `*` eliminates the need to name the application, which is unnecessary because the files are private to *exmh*. This also avoids the naming quirks of Tk applications when there are multiple instances (e.g., "`exmh`" *vs.* "`exmh #2`").

```
puts $prefFile "*$resource: $value"
```

These values are retrieved with the Tk `option` command:

```
proc PrefValue { varName xres } {
    upvar #0 $varName var
    set var [option get . $xres {}]
}
```

4 X resources

The most obvious use of X resources is for specifying widget attributes. All the attributes for a Tk widget can be set via resources as opposed to the command line. In fact, if you want users to be able to control fonts, colors, relief, and other appearance-related attributes, you must not set them directly in Tcl code. Doing so overrides any resource specifications.

The advantage to using resource specifications is that the keys in the database are patterns. With just a few resource entries you can define attributes for all your widgets. For example, in Tk 4.0 it is possible to obtain reverse video on a monochrome display with just two resource specifications:

```
*Background: black
*Foreground: white
```

These are resource class specifications, and the various color attributes (e.g., `activeBackground`) have the appropriate class to make this work out nicely. (Tk 3.6 requires a few more specifications for parts of the scrollbar and scale widgets, but has been cleaned up.).

The main drawback to using X resources is that your application is affected by them indirectly. A stray setting in the users `~/.Xdefaults` file can impact your application, or break it altogether as described in the next section. You can retain some control over this situation by using the priority mechanism in the Tk implementation of the resource database. The initial set of attributes, which come from either the `RESOURCE_MANAGER` property or the user's `~/.Xdefaults` file, are loaded at priority `userDefault`. In *exmh*, the `app-defaults` file is loaded at `startupFile` priority, which is lower, but the user's `~/.exmh-defaults` file is loaded at priority `userDefault`. Because it is at the same priority as `~/.Xdefaults`, and because order matters to Tk, settings there can override those in `~/.Xdefaults`. The user can set resources on the `RESOURCE_MANAGER` property in a clever way to account for per-host or per-display characteristics. Or, she can use the structure provided by the `~/.exmh-defaults`, `~/.exmh-defaults-color`, and `~/.exmh-defaults-mono` files and override specifications that lurk in her `~/.Xdefaults`.

A related disadvantage to the X resources database is there is no real user interface for them except your favorite editor. The preference user interface described in the previous section takes care of this for the preference items, but it does not support customization of widget attributes. A generic browser for widget attributes that saves its results as resource specifications would be an ideal tool for all Tk applications. As a temporary measure, the *exmh* `app-defaults` and `app-defaults-color` files specify several generic widget attributes as a guide to users that want to customize things by hand.

4.1 Fonts

Font choices should be defined with X resources. When doing so, you should provide an underspecified font name so that the X server has more of a chance to find a matching font. For example, the following two resources specify a 12 pixel `courier` font. The first is general, while the second is fully qualified:

```
*courier-medium-r-*-*-12*
-adobe-courier-medium-r-normal--12-120-
  75-75-m-70-iso8859-1
```

One of the most fragile aspects of X is its font handling. A missing font or a bogus font specification is a show-stopper. *exmh* uses a set of routines that provide a thin layer over the basic widget commands that create widgets with a font. A generic version of the procedure is shown below. If the widget creation raises an error, it is retried with an explicit font setting of `fixed`, which is the only font guaranteed to exist.

```
proc FontWidget { args } {
    if [catch $args w] {
       # Delete the font specified in
       # args, if any
       set ix [lsearch $args -font]
       if {$ix >= 0} {
          set args [lreplace \
             $args $ix [expr $ix+1]]
       }
       # This font overrides the
       # resource database
       set w [eval $args {-font fixed}]
    }
    return $w
}
```

5 Personal Tcl code

The *exmh* implementation is split into modules, one module per file. The Tcl `auto_load` facility is used to load the files from a script library. The implementation supports a personal library of Tcl modules, too. The original motivation for the personal library was to let users modify parts of *exmh* without affecting other users at their site. The idea was that a user would copy a module into their personal library and modify it. The personal Tcl library is put onto the `auto_load` path first so that procedures defined there override those defined by the system library.

For example, the buttons and menus were defined in one module through a handful of procedures. It was possible to edit these files to tweak the interface, and the code was simple enough that a newcomer to Tcl could probably handle it. There is a maintenance cost to modifying code, however, and this was rarely done. As described in the next section, however, the personal library turns out to be more useful for grafting on new functionality.

The "module override" approach to customization has its drawbacks. A module can include a dozen or more procedures. If the user wants to change one command in one procedure, they must copy the whole module into their personal Tcl library. This is fine until the next release comes out, at which point they either forget about their customized code and run into errors, or they remember and have to figure out how to merge in their changes. In practice this has not been used much for casual customization. It has, however, proven useful to more serious hackers.

Hook points are more useful. At several key places in the *exmh* implementation, it checks for the definition of a user-supplied procedure, and calls it if it exists. More precisely, it calls all the procedures that match a pattern. For example, the following code is executed just before a mail message is sent. `$draft` is the file containing the message, and `$t` is the text widget that displays it.

```
foreach cmd [info commands \
      Hook_SeditSend*] {
   if [catch {$cmd $draft $t} err] {
      Status "$cmd $err"
   }
}
```

There is one well known hook, `User_Init` (the `User` prefix is historical), that *exmh* always calls. A stub version of `user.tcl` is provided by the main script library, and users are expected to provide their own `User_Init` if they have personal code. This provides a hook into the Tcl script library mechanism so their code gets loaded. They define `User_Init` and all their `Hook*` procedures in `user.tcl`, or they can source other files explicitly from within `User_Init`.

A second hook, `User_Layout`, is called after the interface has been initialized. At this point most of *exmh* has been loaded and the serious hacker could redefine individual procedures. You can do this inside `User_Layout` by sourcing additional files, or by directly defining the Tcl procedures. There is a single global scope for procedures, so there is no problem with redefining a system procedure inside `User_Layout`.

6 User-Defined Buttons and Menus

Almost all the buttons and menus in *exmh* are defined by X resources. John Robert LoVerso encouraged me to do this, and I resisted for some time. After he supplied an implementation for buttons, however, I was convinced, and generalized it to handle menus as well. This approach has let users add or change buttons and menus more readily than when they had to override the whole button module.

The main trick to user defined buttons is that there is no way to enumerate the X resource database. Instead, you must introduce resources that list other resources. For example, the main set of buttons in *exmh* is in a frame that is given the class `Main`. The following resource lists the system-defined buttons contained in this frame:

```
*Main.buttonlist: quit pref alias
```

In turn, each of these buttons is defined with resources like this:

```
*Main.quit.text:Quit
*Main.quit.command:Exmh_Done
*Main.quit.cursor:gumby
*Main.pref.text:Preferences
*Main.pref.command:Preferences_Dialog
*Main.alias.text:Aliases
*Main.alias.command:Aliases_Pref
```

In addition, the implementation also checks for the buttons listed in the `*Main.ubuttonlist` resource. The purpose of `ubuttonlist` is to let users easily add more buttons without worrying about the system-defined buttons. Only if they want to remove a system-defined button do they need to specify `*Main.buttonlist`. The code that uses the resources is simple. (The complexity comes from checking for font errors.)

```
set f [frame .main -class Main]
foreach b [concat \
      [option get $f buttonlist {}] \
      [option get $f ubuttonlist {}]] {
   if [catch {button $f.$b} err] {
      Status "(warning) $err"
      button $f.$b -font fixed
   }
   pack $f.$b -side right
}
```

This system has been generalized by Achim Bohnet to allow for site-defined buttons (`lbuttonlist`) in the `local-app-defaults` file. He also allows for deletion by listing buttons in the `l-buttonlist` and `u-buttonlist` resources. This eliminates the need to override the main buttonlist resource to delete a button, and it makes it easier to maintain customizatations when new releases appear.

User-defined menus are more complex because resources specifications are not directly supported for menu entries. The `menulist` resource is used to enumerate the menus. For each menu the `entrylist` resource enumerate the entries. Again, these have been generalized like the buttonlist resource. Along with `menulist`, there is also `lmenulist`, `umenulist`, `l-menulist`, and `u-menulist` resources, and `entrylist` is similar. For example:

```
*Main.menulist:bind help
*Main.bind.text:Bindings...
*Main.bind.m.entrylist: \
   command sedit compose
*Main.help.text:Help...
*Main.help.m.entrylist:\
   help colorkey faq sep bug reg
```

The additional `.m` component is for the menu widget, which is a child of the menubutton. At this point we must be more creative for the individual menu entries. It is tempting to specify them like this:

```
*Main.bind.m.command.label: Commands
*Main.bind.m.command.command: Bind_Pref
*Main.bind.m.sedit.label: Simple Edit
*Main.bind.m.sedit.command: Sedit_Pref
```

However, while you can define these resources, you cannot retrieve them from the resource database! Tk assumes that `.main.bind.m.command` is a widget, but it is not. `.main.bind.m` is a menu widget, but the individual menu entries are not widgets, so you can not define resources for them. We can, however, define resources for the menu that Tk does not recognize, and then request them explicitly. The following resources are used:

- t_*entry* The type of the *entry*: `command`, `radio`, `check`, `cascade`, or `separator`.
- l_*entry* The label for *entry*.
- c_*entry* The command for *entry*.
- v_*entry* The variable associated with *entry*.
- m_*entry* The menu for the cascade *entry*.

The previous resources are specified like this (the default type is command):

```
*Main.bind.m.l_command: Commands
*Main.bind.m.c_command: Bind_Pref
*Main.bind.m.l_sedit: Simple Edit
*Main.bind.m.c_sedit: Sedit_Pref
```

The code to use these resources is a little more complicated. A procedure is necessary to support the recursion implied by cascade menu types.

```
foreach M [concat \
    [option get $frame menulist {}] \
    [option get $frame umenulist {}]] {
    if [catch {menubutton $f.$M \
          -menu $f.$M.m} err] {
       Status "(warning) $err"
       menubutton $f.$M \
          -menu $f.$M.m -font fixed
    }
    pack $f.$M -side right
    ButtonMenuInner $f.$M.m
}
proc ButtonMenuInner {menu} {
    foreach e [concat \
      [option get $menu entrylist {}] \
      [option get $menu uentrylist {}]] {
       set l [option get $menu l_$e {}]
       set c [option get $menu c_$e {}]
       set v [option get $menu v_$e {}]
       case [option get $menu t_$e {}] {
          check {$menu add check \
    -label $l -command $c -variable $v}
          radio {$menu add radio \
    -label $l -command $c -variable $v}
          cascade {
             set sub [option get \
                $menu m_$e {}]
             if {[string length $sub] \
                   != 0} {
                set submenu \
                      [menu $menu.$sub]
                $menu add cascade \
             -label $l -menu $submenu \
                      -command $c
                ButtonMenuInner $submenu
                }
             }
             separator {$menu add \
                separator}
             default {$menu add command \
                -label $l -command $c}
          }
       }
    }
```

7 Key Bindings

There are two sets of key bindings in *exmh*. One is for keyboard accelerators for commands normally accessed via buttons and menus. These bindings are in effect when the user is reading mail. The other set is for editing text. These bindings are in effect in the mail composition window, and in all entry widgets. Both of these are set up via simple user interfaces.

The editing interface displays about 20 edit functions and lets the user define Tk events (e.g., `<Control-d>`) that invoke them. The functions are mnemonics; they are not raw Tcl commands. They include `backspace`, `forw1char`, `backword`, `down1line`, and so on. The decision to go with mnemonics instead of commands is debatable. I did it because I set up bindings for both the `Text` and `Entry` widget classes, and these have slightly different text operations. The interface hides that problem. However, it makes it harder for users to define custom edit operations. They have to override the `seditBind.tcl` module to add new edit functions.

The command binding interface is more general. It displays a list of Tcl commands and the associated Tk event. Users can change the Tk event for an existing command, and they can add new commands. The default set of Tcl commands available through key bindings are all simple commands like `Folder_Commit`, `Msg_Remove`, and `Inc`. However, the user is free to define a binding for a more complex command:

```
Msg_Reply -cc to -cc cc -filter mhl.reply
```

The drawback with the command binding interface is that it does not help users keep keybindings up-to-date with their button definitions. For *mxedit* I developed a menu package that automatically kept the menu definition, the key binding, and the accelerator display in the menu all up-to-date. It would be ideal if the interface let the user associate a keystroke with a button or menu entry, and only exposed the underlying Tcl command if requested.

8 Experiences with exmh

Exmh has grown considerably since its first release in August, 1993. The growth has been primarily due to user demand. I was the sole user of *exmh* from November, 1992 to August 1993, and it would still be much the same program if I had not let others use it. Real users with real complaints are very motivational.

The extensibility of *exmh* has been very important in its success. The ability to define buttons and menus without touching the main code has been popular. The personal Tcl library has let users add small (and large) amounts of code to exmh. Users have been able to experiment with new features without digging into the main body of code.

I only partly succeeded in my goal of not implementing every requested feature. However, significant chunks of the program were done by users as extensions that I later folded into the main distribution. For example, PGP support was added initially by Allen Downey, tweaked by a few others, and eventually Stefan Monnier rewrote it and now maintains it. The alias browser was contributed by Scott Stanton. The URL highlighting was contributed by Martin Hamilton. John LoVerso contributed the hierarchical folder display and the extensible button package. Chris Garrigues rewrote the MIME display code. Marc VanHeyningen contributed the mailcap parsing package. Achim Bohnet added the local site preferences to the extensible button and menu system. Many others contributed ideas and small fixes. By now the implementation has grown to over 22,000 lines of Tcl in about 70 files.

This growth is supported by a trivial module system. Each file contains a module. For example, `fdisp.tcl` contains the folder display module. Every procedure in a file has a common prefix (e.g., `Fdisp`) to avoid name conflicts with other modules. Procedures that are meant to be called by other modules have an underscore after their prefix (e.g., `Fdisp_Init`). Each module uses a global array to hold its state variables, and its name is the module prefix in lower case (e.g., `fdisp`). These naming conventions were in place before the first release of exmh, and they have helped keep the implementation tidy as it grew by an order of magnitude.

Users appreciate support and active development. *Exmh* is certainly not perfect. However, my response to bug reports is often rapid. Simple fixes are easy in Tcl, and I can mail out patch files that users can readily apply to their sources. This would not be nearly as easy with a compiled program. And if there is a really common bug, I can define a new menu entry in my personal version of exmh that sends out the appropriate reply!

Users have two views about the customization of *exmh*. Many folks do not want to read through 15 sets of preference items and adjust everything just so, nor spend time with colors, fonts, and key bindings. They want it to work "right" the first time. Others enjoy the control. In practice, it appears that one person becomes a local expert and configures *exmh* for others. The three-level scheme to support system defaults, local site defaults, and per user defaults reflects this.

9 Availability

`parcftp.xerox.com:/pub/exmh/exmh-1.6.tar`

This file may also be on `ftp.aud.alcatel.com`, which is the current Tcl archive site. It is usually compressed or gzip'ed, so look for the .Z or .gz file. The version number will continue to change as *exmh* evolves. There are typically several development releases followed by a "stable" release. You can identify the development releases from the alpha, beta, gamma, delta (and so on) qualifier on the version (e.g., 1.7alpha).

10 References

Garfinkle95 Simpson Garfinkle, *PGP: Pretty Good Privacy*, O'Reilly & Associates, Inc. Janurary 1995

Glimpse94 http://glimpse.cs.arizona.edu:1994/

Ousterhout94 John Ousterhout, *Tcl and the Tk Toolkit*, Addison Wesley, April 1994

Peek95 Jerry Peek, *MH & xmh: Email for Users and Programmers*, 3rd Edition, O'Reilly & Associates, Inc. April 1995

Welch95 Brent Welch, *Practical Programming in Tcl and Tk*. Prentice Hall, May 1995.

Customization and Flexibility in the exmh Mail User Interface

Brent Welch
brent.welch@eng.sun.com
Sun Microsystems Laboratories
2550 Garcia Ave. M/S UMTV29-232
Mountain View, CA 94043

Abstract

The exmh mail user interface is designed to be flexible and customizable. The goal in the design is to provide a platform that can be tuned to suit individual preferences, as well as extended to provide custom functionality. There are several mechanisms for customization: install-time settings define runtime support requirements; a Preferences user interface exposes various "knobs" and "dials"; the X resource database controls fonts, colors, and other widget attributes; a personal library of Tcl procedures supports new functionality; the buttons and menus in the interface are defined by X resource specifications to allow customization and access to personal Tcl extensions; hook points in the implementation support callbacks to user extensions at key moments in mail handling; A binding user interface defines editing and accelerator keystrokes. This paper describes these mechanisms, and provides some experiences in the use and development of exmh.

1 Introduction

Exmh is a user interface to MH mail. MH was developed at RAND has been freely available for several years. It is a collection of UNIX programs that manage your mail. MH uses the file system for its state: a mail folder is a directory, each message is a file in a directory, and a few adjunct files maintain profile and context information such as the current folder. Instead of running one monolithic mail program, you can use the MH programs individually from the UNIX command line. Or, you can assemble them back into a monolithic mail reader that is tuned for your needs. The building-block nature of the MH programs make them well suited for composition with a Tcl script and a Tk graphical interface.

Exmh started as a short script I got from Ed Oskiewicz. It provided a starting point as I had not used MH before that time. I debugged, enhanced, and used exmh myself for several months before I decided to give it out to the rest of the Tcl community. I realized that exmh needed to be flexible. After all, the reason I was using it was that I wanted control over my mail reading environment. I also did not want to implement all the features that would inevitably be requested once other folks began using it. There were three means of customization in the initial version: the installer for system dependencies, a preferences package, and support for a library of personal Tcl code. The preference package exposed several "knobs" and "dials" that I thought users might want to change. The Tcl library was set up so users could copy parts of the implementation and fix or enhance them without changing the main application. In support of that goal, I broke the implementation up into several modules instead of having one large script.

The remainder of this paper describes how these customization features evolved, and the new customization features that were added to increase the flexibility of the system. The paper concludes with a few remarks about the experience of developing and maintaining exmh. As of this writing, over 800 different sites pick up each release of exmh, and almost 700 users have taken the time to register as an exmh user. I estimate there are a few thousand *exmh* users in total.

2 Installation and System Dependencies

Exmh is built on top of several packages, including MH [Peek95], Tcl/Tk[Ousterhout94][Welch95], Metamail (for MIME support), PGP (for public key encryption and digital signatures) [Garfinkle95], Glimpse (for full text indexing) [Glimpse94], and the facesaver database. To use these packages, *exmh* must know where they live in the file system. Instead of defining several environment variables that record this information, the locations are patched into *exmh* during installation. This is done with a simple installation program that is written as a Tcl/Tk script. After unpacking the *exmh* tar file, you run its installer like this:

```
% wish -f exmh.install
```

The implementation of the installer is table-driven. The `exmh.install` script consists of a series of directives like this:

```
install_var exmh(maintainer) \
    brent.welch@sun.com \
    {Target of error mailings}
install_dirVar mh_path /usr/local/bin
  {MH binary directory}
```

These are Tcl commands that take three arguments. The first argument is the name of a Tcl variable or array element, the second is its default value, and the third is a short text label for the installer user interface. These commands result in Tcl variable definitions in the final *exmh* script. The `install_var` command is the basic operation, and there are several variations that imply types for the variable values. These include `install_dirVar`, `install_fileVar`, and `install_progVar` that define directories, files, and executable programs, respectively. The install script does some simple checks to ensure that the values are ok.

The files to be installed are defined with two directives. `install_dir` specifies a directory, and `install_glob` defines a file name pattern of sources from the distribution that must be copied to the directory. (These two commands could be combined by adding the pattern to `install_dir`.) For example:

```
install_dir man /usr/man/manl \
    {Install man directory}
install_glob man exmh*.l
```

There are a few more specialized commands used to define help text and set up the testing environment used before the package is installed. Their details are not that important here. The main idea is to capture all the steps of installing a Tcl/Tk program, and to expose any dependencies on other packages to the system administrator.

The install package uses the directives to build a simple user interface that lists each value that must be set. There are buttons labeled `Patch`, `Test`, `Verify`, `Install`, `Config`, and `Quit`. The `Patch` operation uses *sed* to insert the Tcl variable definitions into the main *exmh* script. It is at this time that the semantic checks on the settings are made. Any errors are displayed to the user. This step also fixes the pathname for *wish* that appears in the first line of the script. Even if the install script gets lost, the system dependencies appear clearly marked right at the beginning of the *exmh* script.

The `Test` button runs *exmh* without installing it. This uses the script library as unpacked from the tar file. The `Verify` operation lists the commands that will be issued when the user clicks `Install`.

The `Config` button is used to select an existing set of definitions. A side effect of `Patch` is to save the settings into a `.exmhinstall` file. The installer looks in the current and peer directories for this file. This makes it easy to install new versions if you have installed *exmh* before.

You can pick up the `exmh.install` script and the supporting `install.tcl` Tcl procedures from:

```
ftp://parcftp.xerox.com/pub/exmh/
```

3 The Preferences Package

User preference settings are common in any large application. There are always design decisions that can go either way, and some users prefer it one way while others do not. For example, should “next message” skip over messages that are marked for delete, or should they be displayed? What command should be used to print a message? What editor do you want to use?

The *exmh* preference packages has three types of preference items: booleans, choices, and general strings. In the user interface, a checkbutton represents a boolean, a set of radiobuttons represent a choice, and an entry represents a general string. This is a fairly obvious approach. Sets of preference items are registered together under a heading such as “Background Processing” or “Folder Display”, and the preference interface is composed of one main menu and then a toplevel window for each preference set.

A few issues arose during the design of the preference package:

- Definition of preference items.
- Management of a large number of settings.
- Representation for the settings for long term storage.

3.1 Definition of Preference Items

A set of preference items is registered with the `Pref_Add` command. It takes as arguments the name of the set, some general information, and a list of preference items. Each item is described by a Tcl variable, an X resource name, which is discussed later, an identifying label, and some additional help text. The user interface is generated automatically from this specification.

A call to `Pref_Add` looks like this:

```
Pref_Add title text listOfItems
```

Each item is a list of 5 elements:

```
var xres default label helpText
```

Here is an example:

```
Pref_Add "Background Processing" \
"A second process is used to perform various background tasks for Exmh. These
  options control its behavior." {
{exmh(bgAsync) bgAsync ON
    {Separate background process}
    "This setting determines whether or
  not the background processing is done
  in a separate process or not. A separate process causes less interference
  with the user interface, but might take
  up more machine resources."}
{exmh(background) bgAction
    {CHOICE off count msgchk flist inc}
    {Background processing}
```

```
    "exmh can periodically do some things
   for you:
    count - count new messages sitting in
   your spool file.
    msgchk - run the MH msgchk program.
    flist - check for new mail in all
   folders.
    inc - just like clicking the Inc but-
   ton yourself.
    off - do nothing in the background."}
 {exmh(bgPeriod) bgPeriod 10
    {Period (minutes)}
    "How often to do background task"}
 }
```

3.2 Managing Many Preference Items

The table driven approach to preference items makes it easy to add more. The original implementation did not divide the items into sets. As *exmh* evolved, however, the number of preference items grew large enough to become unwieldy. Among other things, a single window that displayed all the items was too large for some screens! A relatively minor change to the package was required to support a two-level scheme. Each module of the implementation already registered a set of preference items with a single call to `Pref_Add`. Its interface was changed to accept a name for the set and some overall information, and the layout of the display was changed to a menu with separate windows for each set.

The main drawback with the current approach is that it is implementation-based. The sets correspond to *exmh* modules that register preference settings. There are about 15 different preference sets, and it can be hard to find a particular setting. This is not inherent in the preferences package, however. It would be easy to add a search mechanism and an alternate display that showed all the settings in a compact manner. You could also reorganize the calls to `Pref_Add` to reflect the way the user thinks as opposed to the way *exmh* is built.

3.3 Storing Preference Data

The first version of the preference package saved the settings as a set of Tcl commands that initialized variables. The only trick is the proper formatting of the commands. Assuming the name of the preference variable is in `$varName`, the following commands work. The `list` command is essential to allow for arbitrary values:

```
upvar #0 $varName value
puts $prefFile \
    [list set $varName $value]
```

However, as described in the next section, *exmh* is also affected by a set of per-user X resource specifications for fonts, colors and other widget attributes. It seemed odd to users to have a `~/.exmhpref` for preference settings and a `~/.exmh-defaults` for resources. I was reluctant to drop the Tcl commands approach, but I finally gave in to user demand.

The default settings for *exmh* are stored in `app-defaults`, which is kept in the script library. Settings that are particular to a color display are in `app-defaults-color`, and settings for black and white displays are in `app-defaults-mono`. Personal settings mirror this arrangement. The `~/.exmh-defaults` file has preference settings and other generic resources. `~/.exmh-defaults-color` and `~/.exmh-defaults-mono` are used for color and monochrome display settings, respectively. Recently an additional set of files were added, `local-app-defaults`, `local-app-defaults-color`, and `local-app-defaults-mono`. This is for site-specific settings so that administrators can maintain a standard set of preference settings without modifying `app-defaults`. The three sets of files correspond to “factory settings”, “site settings”, and “personal settings”. Tk associates a priority with resources, and the personal settings have the highest priority while the factory settings have the lowest priority.

When writing out the preference settings as resource values there is no need to worry about quoting. However, there is no support for resource values that contain leading blanks. The X resource database file format does not support it, and I did not invent any special coding for it.

The following commands generate a resource file specification. The leading `*` eliminates the need to name the application, which is unnecessary because the files are private to *exmh*. This also avoids the naming quirks of Tk applications when there are multiple instances (e.g., “`exmh`” *vs.* “`exmh #2`”).

```
puts $prefFile "*$resource: $value"
```

These values are retrieved with the Tk `option` command:

```
proc PrefValue { varName xres } {
    upvar #0 $varName var
    set var [option get . $xres {}]
}
```

4 X resources

The most obvious use of X resources is for specifying widget attributes. All the attributes for a Tk widget can be set via resources as opposed to the command line. In fact, if you want users to be able to control fonts, colors, relief, and other appearance-related attributes, you must not set them directly in Tcl code. Doing so overrides any resource specifications.

The advantage to using resource specifications is that the keys in the database are patterns. With just a few resource entries you can define attributes for all your widgets. For example, in Tk 4.0 it is possible to obtain reverse video on a monochrome display with just two resource specifications:

```
*Background: black
*Foreground: white
```

These are resource class specifications, and the various color attributes (e.g., `activeBackground`) have the appropriate class to make this work out nicely. (Tk 3.6 requires a few more specifications for parts of the scrollbar and scale widgets, but has been cleaned up.).

The main drawback to using X resources is that your application is affected by them indirectly. A stray setting in the users `~/.Xdefaults` file can impact your application, or break it altogether as described in the next section. You can retain some control over this situation by using the priority mechanism in the Tk implementation of the resource database. The initial set of attributes, which come from either the `RESOURCE_MANAGER` property or the user's `~/.Xdefaults` file, are loaded at priority `userDefault`. In *exmh*, the `app-defaults` file is loaded at `startupFile` priority, which is lower, but the user's `~/.exmh-defaults` file is loaded at priority `userDefault`. Because it is at the same priority as `~/.Xdefaults`, and because order matters to Tk, settings there can override those in `~/.Xdefaults`. The user can set resources on the `RESOURCE_MANAGER` property in a clever way to account for per-host or per-display characteristics. Or, she can use the structure provided by the `~/.exmh-defaults`, `~/.exmh-defaults-color`, and `~/.exmh-defaults-mono` files and override specifications that lurk in her `~/.Xdefaults`.

A related disadvantage to the X resources database is there is no real user interface for them except your favorite editor. The preference user interface described in the previous section takes care of this for the preference items, but it does not support customization of widget attributes. A generic browser for widget attributes that saves its results as resource specifications would be an ideal tool for all Tk applications. As a temporary measure, the *exmh* `app-defaults` and `app-defaults-color` files specify several generic widget attributes as a guide to users that want to customize things by hand.

4.1 Fonts

Font choices should be defined with X resources. When doing so, you should provide an underspecified font name so that the X server has more of a chance to find a matching font. For example, the following two resources specify a 12 pixel `courier` font. The first is general, while the second is fully qualified:

```
*courier-medium-r-*-*-12*
-adobe-courier-medium-r-normal--12-120-
  75-75-m-70-iso8859-1
```

One of the most fragile aspects of X is its font handling. A missing font or a bogus font specification is a show-stopper. *exmh* uses a set of routines that provide a thin layer over the basic widget commands that create widgets with a font. A generic version of the procedure is shown below. If the widget creation raises an error, it is retried with an explicit font setting of `fixed`, which is the only font guaranteed to exist.

```
proc FontWidget { args } {
    if [catch $args w] {
        # Delete the font specified in
        # args, if any
        set ix [lsearch $args -font]
        if {$ix >= 0} {
            set args [lreplace \
                $args $ix [expr $ix+1]]
        }
        # This font overrides the
        # resource database
        set w [eval $args {-font fixed}]
    }
    return $w
}
```

5 Personal Tcl code

The *exmh* implementation is split into modules, one module per file. The Tcl `auto_load` facility is used to load the files from a script library. The implementation supports a personal library of Tcl modules, too. The original motivation for the personal library was to let users modify parts of *exmh* without affecting other users at their site. The idea was that a user would copy a module into their personal library and modify it. The personal Tcl library is put onto the `auto_load` path first so that procedures defined there override those defined by the system library.

For example, the buttons and menus were defined in one module through a handful of procedures. It was possible to edit these files to tweak the interface, and the code was simple enough that a newcomer to Tcl could probably handle it. There is a maintenance cost to modifying code, however, and this was rarely done. As described in the next section, however, the personal library turns out to be more useful for grafting on new functionality.

The “module override” approach to customization has its drawbacks. A module can include a dozen or more procedures. If the user wants to change one command in one procedure, they must copy the whole module into their personal Tcl library. This is fine until the next release comes out, at which point they either forget about their customized code and run into errors, or they remember and have to figure out how to merge in their changes. In practice this has not been used much for casual customization. It has, however, proven useful to more serious hackers.

Hook points are more useful. At several key places in the *exmh* implementation, it checks for the definition of a user-supplied procedure, and calls it if it exists. More precisely, it calls all the procedures that match a pattern. For example, the following code is executed just before a mail message is sent. `$draft` is the file containing the message, and `$t` is the text widget that displays it.

```
foreach cmd [info commands \
      Hook_SeditSend*] {
   if [catch {$cmd $draft $t} err] {
      Status "$cmd $err"
   }
}
```

There is one well known hook, `User_Init` (the `User` prefix is historical), that *exmh* always calls. A stub version of `user.tcl` is provided by the main script library, and users are expected to provide their own `User_Init` if they have personal code. This provides a hook into the Tcl script library mechanism so their code gets loaded. They define `User_Init` and all their `Hook*` procedures in `user.tcl`, or they can source other files explicitly from within `User_Init`.

A second hook, `User_Layout`, is called after the interface has been initialized. At this point most of *exmh* has been loaded and the serious hacker could redefine individual procedures. You can do this inside `User_Layout` by sourcing additional files, or by directly defining the Tcl procedures. There is a single global scope for procedures, so there is no problem with redefining a system procedure inside `User_Layout`.

6 User-Defined Buttons and Menus

Almost all the buttons and menus in *exmh* are defined by X resources. John Robert LoVerso encouraged me to do this, and I resisted for some time. After he supplied an implementation for buttons, however, I was convinced, and generalized it to handle menus as well. This approach has let users add or change buttons and menus more readily than when they had to override the whole button module.

The main trick to user defined buttons is that there is no way to enumerate the X resource database. Instead, you must introduce resources that list other resources. For example, the main set of buttons in *exmh* is in a frame that is given the class `Main`. The following resource lists the system-defined buttons contained in this frame:

```
*Main.buttonlist: quit pref alias
```

In turn, each of these buttons is defined with resources like this:

```
*Main.quit.text:Quit
*Main.quit.command:Exmh_Done
*Main.quit.cursor:gumby
*Main.pref.text:Preferences
*Main.pref.command:Preferences_Dialog
*Main.alias.text:Aliases
*Main.alias.command:Aliases_Pref
```

In addition, the implementation also checks for the buttons listed in the `*Main.ubuttonlist` resource. The purpose of `ubuttonlist` is to let users easily add more buttons without worrying about the system-defined buttons. Only if they want to remove a system-defined button do they need to specify `*Main.buttonlist`. The code that uses the resources is simple. (The complexity comes from checking for font errors.)

```
set f [frame .main -class Main]
foreach b [concat \
      [option get $f buttonlist {}] \
      [option get $f ubuttonlist {}]] {
   if [catch {button $f.$b} err] {
      Status "(warning) $err"
      button $f.$b -font fixed
   }
   pack $f.$b -side right
}
```

This system has been generalized by Achim Bohnet to allow for site-defined buttons (`lbuttonlist`) in the `local-app-defaults` file. He also allows for deletion by listing buttons in the `l-buttonlist` and `u-buttonlist` resources. This eliminates the need to override the main buttonlist resource to delete a button, and it makes it easier to maintain customizatations when new releases appear.

User-defined menus are more complex because resources specifications are not directly supported for menu entries. The `menulist` resource is used to enumerate the menus. For each menu the `entrylist` resource enumerate the entries. Again, these have been generalized like the buttonlist resource. Along with `menulist`, there is also `lmenulist`, `umenulist`, `l-menulist`, and `u-menulist` resources, and `entrylist` is similar. For example:

```
*Main.menulist:bind help
*Main.bind.text:Bindings...
*Main.bind.m.entrylist: \
   command sedit compose
*Main.help.text:Help...
*Main.help.m.entrylist:\
   help colorkey faq sep bug reg
```

The additional `.m` component is for the menu widget, which is a child of the menubutton. At this point we must be more creative for the individual menu entries. It is tempting to specify them like this:

```
*Main.bind.m.command.label: Commands
*Main.bind.m.command.command: Bind_Pref
*Main.bind.m.sedit.label: Simple Edit
*Main.bind.m.sedit.command: Sedit_Pref
```

However, while you can define these resources, you cannot retrieve them from the resource database! Tk assumes that `.main.bind.m.command` is a widget, but it is not. `.main.bind.m` is a menu widget, but the individual menu entries are not widgets, so you can not define resources for them. We can, however, define resources for the menu that Tk does not recognize, and then request them explicitly. The following resources are used:

- t_*entry* The type of the *entry*: `command`, `radio`, `check`, `cascade`, or `separator`.
- l_*entry* The label for *entry*.
- c_*entry* The command for *entry*.
- v_*entry* The variable associated with *entry*.
- m_*entry* The menu for the cascade *entry*.

The previous resources are specified like this (the default type is command):

```
*Main.bind.m.l_command: Commands
*Main.bind.m.c_command: Bind_Pref
*Main.bind.m.l_sedit: Simple Edit
*Main.bind.m.c_sedit: Sedit_Pref
```

The code to use these resources is a little more complicated. A procedure is necessary to support the recursion implied by cascade menu types.

```
foreach M [concat \
    [option get $frame menulist {}] \
    [option get $frame umenulist {}]] {
    if [catch {menubutton $f.$M \
          -menu $f.$M.m} err] {
       Status "(warning) $err"
       menubutton $f.$M \
          -menu $f.$M.m -font fixed
    }
    pack $f.$M -side right
    ButtonMenuInner $f.$M.m
}
proc ButtonMenuInner {menu} {
    foreach e [concat \
      [option get $menu entrylist {}] \
      [option get $menu uentrylist {}]] {
       set l [option get $menu l_$e {}]
       set c [option get $menu c_$e {}]
       set v [option get $menu v_$e {}]
       case [option get $menu t_$e {}] {
          check {$menu add check \
    -label $l -command $c -variable $v}
          radio {$menu add radio \
    -label $l -command $c -variable $v}
          cascade {
             set sub [option get \
                $menu m_$e {}]
             if {[string length $sub] \
                   != 0} {
                set submenu \
                      [menu $menu.$sub]
                $menu add cascade \
             -label $l -menu $submenu \
                      -command $c
                ButtonMenuInner $submenu
                }
             }
             separator {$menu add \
                separator}
             default {$menu add command \
                -label $l -command $c}
          }
       }
    }
```

7 Key Bindings

There are two sets of key bindings in *exmh*. One is for keyboard accelerators for commands normally accessed via buttons and menus. These bindings are in effect when the user is reading mail. The other set is for editing text. These bindings are in effect in the mail composition window, and in all entry widgets. Both of these are set up via simple user interfaces.

The editing interface displays about 20 edit functions and lets the user define Tk events (e.g., `<Control-d>`) that invoke them. The functions are mnemonics; they are not raw Tcl commands. They include `backspace`, `forw1char`, `backword`, `down1line`, and so on. The decision to go with mnemonics instead of commands is debatable. I did it because I set up bindings for both the `Text` and `Entry` widget classes, and these have slightly different text operations. The interface hides that problem. However, it makes it harder for users to define custom edit operations. They have to override the `seditBind.tcl` module to add new edit functions.

The command binding interface is more general. It displays a list of Tcl commands and the associated Tk event. Users can change the Tk event for an existing command, and they can add new commands. The default set of Tcl commands available through key bindings are all simple commands like `Folder_Commit`, `Msg_Remove`, and `Inc`. However, the user is free to define a binding for a more complex command:

```
Msg_Reply -cc to -cc cc -filter mhl.reply
```

The drawback with the command binding interface is that it does not help users keep keybindings up-to-date with their button definitions. For *mxedit* I developed a menu package that automatically kept the menu definition, the key binding, and the accelerator display in the menu all up-to-date. It would be ideal if the interface let the user associate a keystroke with a button or menu entry, and only exposed the underlying Tcl command if requested.

8 Experiences with exmh

Exmh has grown considerably since its first release in August, 1993. The growth has been primarily due to user demand. I was the sole user of *exmh* from November, 1992 to August 1993, and it would still be much the same program if I had not let others use it. Real users with real complaints are very motivational.

The extensibility of *exmh* has been very important in its success. The ability to define buttons and menus without touching the main code has been popular. The personal Tcl library has let users add small (and large) amounts of code to exmh. Users have been able to experiment with new features without digging into the main body of code.

I only partly succeeded in my goal of not implementing every requested feature. However, significant chunks of the program were done by users as extensions that I later folded into the main distribution. For example, PGP support was added initially by Allen Downey, tweaked by a few others, and eventually Stefan Monnier rewrote it and now maintains it. The alias browser was contributed by Scott Stanton. The URL highlighting was contributed by Martin Hamilton. John LoVerso contributed the hierarchical folder display and the extensible button package. Chris Garrigues rewrote the MIME display code. Marc VanHeyningen contributed the mailcap parsing package. Achim Bohnet added the local site preferences to the extensible button and menu system. Many others contributed ideas and small fixes. By now the implementation has grown to over 22,000 lines of Tcl in about 70 files.

This growth is supported by a trivial module system. Each file contains a module. For example, `fdisp.tcl` contains the folder display module. Every procedure in a file has a common prefix (e.g., `Fdisp`) to avoid name conflicts with other modules. Procedures that are meant to be called by other modules have an underscore after their prefix (e.g., `Fdisp_Init`). Each module uses a global array to hold its state variables, and its name is the module prefix in lower case (e.g., `fdisp`). These naming conventions were in place before the first release of exmh, and they have helped keep the implementation tidy as it grew by an order of magnitude.

Users appreciate support and active development. *Exmh* is certainly not perfect. However, my response to bug reports is often rapid. Simple fixes are easy in Tcl, and I can mail out patch files that users can readily apply to their sources. This would not be nearly as easy with a compiled program. And if there is a really common bug, I can define a new menu entry in my personal version of exmh that sends out the appropriate reply!

Users have two views about the customization of *exmh*. Many folks do not want to read through 15 sets of preference items and adjust everything just so, nor spend time with colors, fonts, and key bindings. They want it to work "right" the first time. Others enjoy the control. In practice, it appears that one person becomes a local expert and configures *exmh* for others. The three-level scheme to support system defaults, local site defaults, and per user defaults reflects this.

9 Availability

`parcftp.xerox.com:/pub/exmh/exmh-1.6.tar`

This file may also be on `ftp.aud.alcatel.com`, which is the current Tcl archive site. It is usually compressed or gzip'ed, so look for the .Z or .gz file. The version number will continue to change as *exmh* evolves. There are typically several development releases followed by a "stable" release. You can identify the development releases from the alpha, beta, gamma, delta (and so on) qualifier on the version (e.g., 1.7alpha).

10 References

Garfinkle95 Simpson Garfinkle, *PGP: Pretty Good Privacy*, O'Reilly & Associates, Inc. Janurary 1995

Glimpse94 http://glimpse.cs.arizona.edu:1994/

Ousterhout94 John Ousterhout, *Tcl and the Tk Toolkit*, Addison Wesley, April 1994

Peek95 Jerry Peek, *MH & xmh: Email for Users and Programmers*, 3rd Edition, O'Reilly & Associates, Inc. April 1995

Welch95 Brent Welch, *Practical Programming in Tcl and Tk*. Prentice Hall, May 1995.