Mega-widgets in Tcl/Tk:
Evaluation and Analysis

Shannon Jaeger

Department of Computer Science
University of Calgary
Calgary, Alberta, Canada T2N 1N4
jaeger@cpsc.ucalgary.ca

This paper presents a framework for evaluating Tk mega-widget extensions. This framework addresses how these extensions perform, both from the application builder's view of created widgets as well as from the viewpoint of the mega-widget builder. Issues addressed include support for building Tk-like widgets, access to component widgets, and reuse of previous widget implementations. Several existing mega-widget extensions are then evaluated using the framework.

1 Introduction

McLennan defined a *mega-widget* as "a collection of primitive widgets [packaged] together as a new widget" [3]. Because mega-widgets are presented as single widgets, they are far easier for Tk developers to use than having to program many individual components. For example, Figure 1 shows a `viewport` mega-widget, created using the Wigwam mega-widget extension. It consists of three component widgets: a horizontal scrollbar, a vertical scrollbar and a viewing widget (any scrollable widget containing text or graphics). This widget provides a great deal of programming flexibility since the scrollbars may or may not be shown, and their placement (top, bottom, left or right) can be altered. The figure shows one particular configuration, where the viewing widget is a listbox with a grocery list and the scrollbars are placed on the left and the bottom. Other common examples of mega-widgets are combo boxes and file browsers.

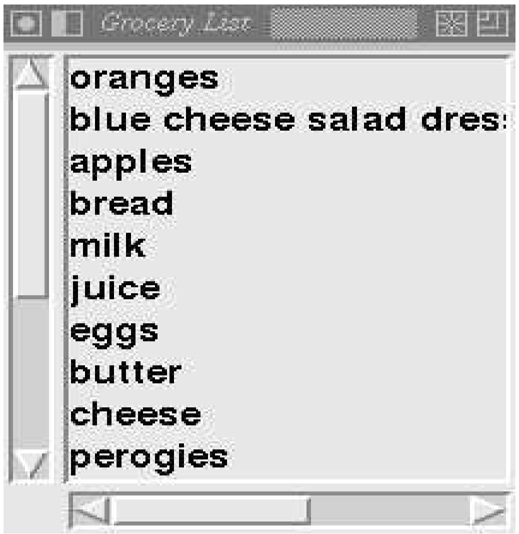

Figure 1: The Viewport Mega-widget

Tk has no direct support for building mega-widgets. As a result, a variety of people have developed extensions for mega-widget construction. These extensions vary greatly in the features they provide, as well as the ways they allow mega-widgets to be constructed. Are all of these features really needed? Are some features missing? Is the programming approach appropriate for the widget creator? Clearly, it is time to consider what developers really require when constructing and using mega-widgets.

This paper presents a framework that considers the needs of developers, which can be used to evaluate existing and future mega-widget systems. The framework considers the different features a constructed mega-widget should support, and the ways in which the programming paradigm should ease a developer's chores. For example, like any widget, a mega-widget will need to worry about handling configuration options such as `-foreground`. A mega-widget, however, also needs to worry about how to propagate this option down to its component widgets. Thus, any mega-widget extension must at the very least permit the widget builder to define how such an option propagates.

The evaluation framework is then used to compare several of the existing mega-widget extensions, as well as more general object-oriented extensions which make some claim to supporting mega-widget development. Whenever possible, the evaluation is based on our actual experience with the extension, although we have also relied on examination of the source code, examples and written documentation.

The evaluation highlights the variety of options available in current mega-widget extensions and common methodologies, as well as areas that remain poorly supported. This information will be useful as the Tcl community begins to converge on a "standard" mega-widget model, as it not only addresses features of mega-widgets, but identifies deficiencies in the Tk core.

2 Evaluation Framework

This section describes the evaluation framework, which contains two parts: the application builder's view and the widget builder's view.

The application builder is the person programming a Tcl/Tk application that contains mega-widgets. They view mega-widgets indirectly; their concern is whether or not a mega-widget behaves "properly." In contrast, the widget builder is the person developing the mega-widgets. The widget builder is directly concerned with the facilities provided by the mega-widget extensions to make building mega-widgets a reasonable chore. These different perspectives are depicted in Figure 2.

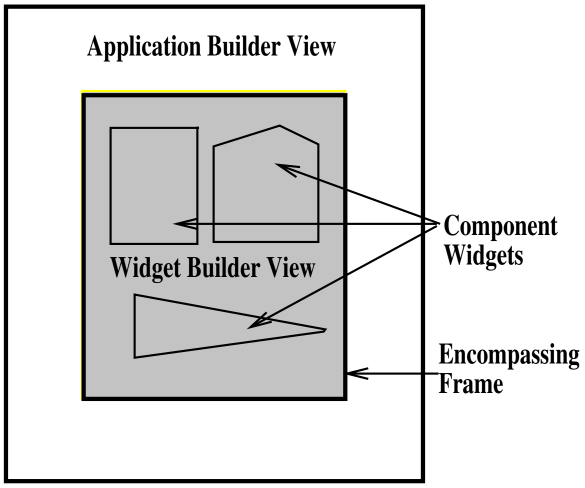

Figure 2: The different views of the Mega-widget

This section begins by reviewing the application builder's view of the essential components in Tk widgets, as well as what mega-widgets should offer. It continues by raising concerns from a widget developer's point of view, including issues such as reuse and namespace conflicts.

2.1 Application Builder View

Mega-widgets should behave like standard Tk widgets. This is desirable to maintain consistency and decrease learning time for the application builder. To make this possible, the mega-widget extension should support the ability to easily create widgets that look and feel "correct" to their users. A secondary consideration for the application builder is the ability — under rare circumstances — to "get inside" the mega-widget to access individual components.

2.1.1 Standard Tk Widget Behaviour

Tk widgets are a set of ready-made controls with a Motif look and feel [5] . Some examples include the button, listbox, text, scrollbar and menu widgets. Application builders access these widgets through a set of properties, described below from a functional point of view. They include the widget creation command, configuration options, the widget command and usage by other Tk commands.

Widget Creation Command. The widget creation command defines new widgets of a given type or widget class. It also creates the widget command and applies the initial set of configuration options. For example, `button .b -foreground red` uses the widget creation command `button` which will create the widget, the command `.b`, and set the button's foreground colour to red.

Widget Command. This command has the same name as the widget's path name, and is created by the widget creation command. It is the main communication link between the Tcl/Tk script and the widget itself. Configurable options and subcommands are changed and executed, respectively, via the widget command procedure. The behaviour of the widget command is dictated by the type of widget.

Configuration Options. All widgets of the same class support the same configurable options. They typically are used to view or change parts of the widget's state information, such as the foreground colour, width, and border type. Another common use for the configurable options is to establish event handlers. Two examples are the listbox `yview` option and the button `command` option.

Widget Subcommands. Widgets may have several subcommands that invoke operations on the widget. For example, all button widgets can process a `flash` subcommand which makes the button flash. Another example is the `configure` subcommand which is responsible for listing and changing the configurable options for each Tk widget.

Bindings. Many widgets have default reactions to specific input events. Application builders can change a widget's behaviour by specifying event bindings with the `bind` command. An example of a default binding occurs when mouse button 1 is pressed and later released over a button widget, causing a command (the value of the `command` option) to be invoked.

Usage with Other Tk Commands. Several Tk commands take widgets as their arguments; they should behave "correctly", yielding similar results when standard Tk widgets and mega-widgets are used with the command. These commands include `bind`, `destroy`, `focus`, `grab`, `lower`, `pack`, `place`, `raise` and `winfo`.

2.1.2 Mega-Widget Components

Ideally, a good mega-widget will directly supply all the functionality and flexibility needed by its users. We realize this is difficult, and that occasionally users will have needs not met directly by the mega-widget. In these cases it would be useful to allow the application builders to "get inside" the mega-widget, and perform operations (invoke commands, change bindings, etc.) directly on a mega-widget's components.

Thus, we believe it would be useful for a mega-widget to provide access to its components, albeit in a controlled way. Ideally a mechanism (e.g. a `component` subcommand) would be provided which takes an "abstract" name and returns the corresponding window path name. For example, a file browser may map the abstract name `filelist` to the path name of a listbox containing the files in the current directory. Using abstract names serves to hide implementation details that are likely to change, while partially exposing relatively static parts of the implementation.

Exposing components in this way is a necessary tradeoff between reuse and the application builder's frustration at not being able to make small but necessary changes in exceptional circumstances. However, a mega-widget whose users are forced to rely frequently on this facility shows evidence of poor design.

2.2 Widget Builder View

The widget builder is responsible for constructing a mega-widget, requiring a view of the extension that is different from that of the user of a mega-widget. Of course, the widget builder needs to consider how well the mega-widget extension supports the application builder's view. Additional concerns are how widgets can be reused, namespace conflicts, creation of top-level widgets and how the extension is actually installed. These issues are discussed below.

2.2.1 Supporting the Application Builder View

The most important offering of a mega-widget development environment is how well it helps the developer construct a widget that supports the application builder's view. To truly support mega-widgets, we feel it is essential to supply all of the standard Tk properties: widget creation command, widget command, options, and subcommands. It would be ideal for the widget builder if many of these were created automatically by the extension and if some of the basics were defined, such as parsing of configuration options. An extension taking care of such "housekeeping chores" can make the job of the widget builder considerably easier.

Widget Creation Command. As described earlier, this command is responsible for creating the widget, creating the widget command, and parsing initial configuration options. We believe that the mega-widget extension should automatically create this command. This requires defining a procedure (whose name is the same as the widget type) that processes configuration options, creates the widget command, and creates the encompassing window which will contain the mega-widget's components. The widget builder then creates and places the individual components and performs any other initializations.

• Encompassing Window. The encompassing window is the widget that all of the component widgets are placed in. This window should be automatically created and the widget builder should be able to specify what type of widget it is. However, the class of the window must be defined properly; its class must be the "mega-widget" class being created, not the type of the window.

Widget Command. This command is responsible for parsing and evaluating widget subcommands. An extension could aid the mega-widget builder by automatically creating such a procedure with some well-defined handling of subcommands and configuration options, although the widget builder should be able to override it if necessary.

• Automatically Creating the Widget Command. The widget builder needs to create the widget command for each and every widget, making it a prime candidate for automation. The command should have the same name as the path name of the mega-widget it is being created for. For example, if the request is to make a combo box widget with the path name `.combo1` then the procedure's name is `.combo1`. It should also provide some high-level mechanism for handling subcommands and options.

Subcommands. Subcommands are operations that can be applied to a particular widget or mega-widget. Some examples are the canvas's `create` command and the button's `activate` and `deactivate` commands. All widgets of the same type have the same subcommands.

• Defining New Subcommands. An important property is the ability to redefine new subcommands for a widget. Ideally, defining the subcommand name and its behaviour should be no more difficult than defining an ordinary Tcl `proc`.

• Automatically Parsing Subcommands. When a widget command is invoked, the correct subcommand must be applied. This is achieved by parsing the arguments to determine what subcommand, if any, is actually being requested. For example, `.listbox insert 0 {Hello World}` inserts the text "Hello World" into a listbox. Here the arguments are `insert 0 {Hello World}` and the subcommand is `insert`. Since this parsing is required for every mega-widget, automatic parsing of subcommands is an ideal candidate for automation by the extension.

• Fallback Behaviour. This is some sort of well-defined behaviour that the widget command implements if nothing is specified by the widget builder. For example, mega-widgets created by an extension may, by default, implement the same standard Tk configuration options and subcommands as the frame widget. The fallback behaviour should also control some of the error detection and notification. One such possible error is using an invalid subcommand with a particular widget.

Configuration Options. Configuration options are part of the widget's state information, such as the foreground colour, width, and border width. Each widget of a particular type has the same options, but widgets of different types may have different options. We've found that dealing with configuration options correctly can be very time consuming for widget builders.

• Defining New Configurable Options. The widget builder should be able to define new configurable options for a particular mega-widget. This allows mega-widgets to be extended by increasing the state information, thus adding more functionality. The viewport widget described earlier has an additional option, `scroll`, which defines where the scrollbars are to be placed. Another example would be a `reverse` option which would reverse the foreground and background colours.

• Defining Option Handlers. This is necessary if new configurable options are allowed. It allows specified handlers for a given option(s). One approach is to allow the widget builder to define a configuration routine that handles all of the options. This technique is useful when a number of options are to be treated in a similar manner. The second method is to define a separate handler for each option. This is also useful, especially when there are only a few options that require "exceptional" handling. Ideally some combination of both methods should be available.

• Automatically Parsing Options. As with subcommands, an extension may eliminate much "housekeeping work" for the mega-widget builder by automatically parsing configuration options. Ideally, configuration options could be "registered" with the mega-widget extension along with code to invoke when the option changes, and the extension would take care of the rest.

• Propagation of Option Changes. A common operation on mega-widgets is to propagate option changes down to the component widgets. An extension can help by allowing a widget builder to specify how changes propagate. For instance, if the background colour is changed for the mega-widget, the builder could specify what component widgets will change their background colour. There are three useful ways that a mega-widget extension can support this:

1. Manual propagation is the simplest approach, requiring the widget builder to deal with the propagation. The option handler manually applies the option to component widgets.

2. Automatic propagation is a more sophisticated approach, allowing the widget builder to specify a list of component widgets an option applies to. Alternatively, a list of options can be specified for each component widget. Then, when a configuration option is changed, the extension automatically propagates the change to the appropriate component widgets.

3. Renaming options when propagating is the ability to map one option to another during propagation, offering fine-grained control. For example, this allows a filebrowser mega-widget to specify a `-listbg` option that is automatically propagated to its listbox component as a `-bg` option.

2.2.2 Reuse

Being able to reuse previously defined widgets promises the benefits of easier debugging, reduced programming time, and more easily maintained programs. Reuse means being able to specify a new mega-widget in terms of existing widgets or mega-widgets.

By definition, mega-widgets support one form of reuse: composition. That is, mega-widgets are created by composing (reusing) other widgets. This is different from the type of reuse where a mega-widget is created by changing or extending an existing widget.

Reuse in the composition sense is usually specified as an extension of the object-oriented metaphor that defines Tk commands. Mega-widget types are analogous to object classes, and changing or extending a widget without composing it into another widget is analogous to creating a subclass of the original widget that inherits all the original's behaviours.

Although a recent discussion on `comp.lang.tcl` about the merits of object-oriented inheritance for building mega-widgets reminds us that the debate is far from resolved, we use the terminology of inheritance here.

Reuse of Existing Tk Widgets. One consideration is the type of widgets that can be reused. One set of widgets that would be useful to reuse are the core Tk widgets. An extension will be more valuable if it allows reuse of these widgets, and not just mega-widgets created with the extension.

Inheriting Subcommands. The ability to inherit subcommands saves the widget builder from redefining them. However, the builder should be able to redefine subcommands as well as access the original ones. For instance, the builder may want to display a message on the screen when a particular subcommand is invoked — this would require redefining the subcommand to first display the message and then invoke the original.

Inheriting Configuration Options. The ability to inherit options saves the builder from redefining these options over and over again. As with subcommands, the builder should be free to redefine, yet have access to the original options and their handlers.

Reuse With Any Encompassing Widget. Several of the extensions automatically create an encompassing frame for the mega-widget. This does not lend itself well to reuse in the form of extending or changing an existing widget; it encourages the placement of a base-level widget within a number of frame widgets due to multiple redefinitions and/or extensions to a base-level widget. The encompassing widget should be allowed to be any valid widget type, since this allows changes to the base-level widget rather than composing it.

2.2.3 Miscellaneous

Other considerations are the creation of top-level widgets, how the extension deals with the namespace problem, automatic option database handling, and if the extension is installed in a standard manner.

Top-Level Widget Support. It is definitely useful to allow the creation of top-level mega-widgets, rather than creating a "normal" widget and then composing it inside of a top-level window. This allows a mega-widget to be a top-level, separate window, rather than something inside of a top-level widget. This is an important feature that shouldn't be overlooked by extensions.

Namespace Support. Mega-widgets contain internal state, both in terms of configuration options as well as code written to support the mega-widget. Extensions can help reduce the conflicts between names used for internal state information and the global information space via some sort of namespace mechanism to provide appropriate scoping [2].

Automatic Database Handling. This is an important feature since it allows a quick method of changing default values for a particular widget type. For example, `option add Viewport*bg red` should set the background colour of all viewport widgets to red. This is similar to, but not the same as, reuse of widgets by changing base-widgets.

"Standard" Installation. Mega-widget extensions (like other Tcl extensions) should be installable in a standard way (e.g. using GNU *autoconf*), and not require complex installation, modifications to core facilities, or make assumptions on where the installation will be.

3 Extension Evaluation

The above criteria were used to evaluate six Tcl/Tk extensions. The results from the evaluation highlight their successes and failures. In order to aid in this assessment a brief description of the extensions is given, followed by tables that rate the various mega-widget extensions.

3.1 Extension Description

The six extensions evaluated are [incr Tcl], Wigwam, [incr Tk], Tix, TkMegaWidget, and theObjects. Table 1 provides detailed information on these extensions including the version examined, the implementation language, the designer(s) and the basis for evaluation. The Y/N values in the *Paper*, *Tested*, and *Code* fields indicate if a paper was read, if widgets were designed in it, and if the code was examined, respectively.

	Version	Language	Paper	Tested	Code
[incr Tcl] *M. McLennan*	1.5	C	Y	Y	Y
Wigwam *J. Wight* *L. Marshall*	1.5b	[incr Tcl]	N	Y	Y
[incr Tk] *M. McLennan*	?	?	Y	N	N
Tix *I. K. Lam*	3.6d	Tcl/Tk	N	Y	Y
TkMegaWidget *S. Delmas*	3.6	C	Y	Y	Y
theObjects *J. Wagner*	3.1	C	N	N	Y

Table 1: Summary of Extension Languages

The focus of some of the extensions is not the support of mega-widgets. For instance, [incr Tcl] is intended as a general-purpose object-oriented extension of Tcl. Wigwam extends [incr Tcl] by adding a set of inheritable classes for the standard Tk widgets. [incr Tk] also extends [incr Tcl], by adding support for building mega-widgets. Tix is designed more from a procedural point of view, and its main purpose is to provide complex widgets. Tix is also the only extension written entirely in Tcl/Tk. The latest version of Tix, which was not examined here, has eliminated some of its shortcomings and is now written in C. TkMegaWidget is designed to make building mega-widgets easier and allows modifying subcommands and options on a per-widget basis [1]. theObjects is a prototype-based object extension, which has been used to create a number of mega-widgets.

3.2 Evaluation Summary

The evaluation summary is presented in two tables: Table 2 details the application builder's view and Table 3, the widget builder's view.

Application Builder View	[incr Tcl]	Wigwam	[incr Tk]	Tix	TkMegaWidget	theObjects
Standard Tk widget behaviour						
Widget creation command	S	S	S	S	S	S
Widget command	S	S	S	S	S	S
Configuration options	S	S	S	S	S	S
Widget subcommands	S	S	S	S	S	S
Bindings	P	P	P	P	P	P
Usage with other Tk commands	P	P	P	P	P	P
Mega-widget behaviour						
Access to component widgets	S	S	S	S	P	P

S supports
D doesn't support
P possibly supports

Table 2: Application Builder View

Widget Builder View	[incr Tcl]	Wigwam	[incr Tk]	Tix	TkMegaWidget	theObjects
Supporting the application builder view						
Widget creation command	+	+	+	+	+	+
encompassing window	○	○	○	+	+	○
Widget command						
automatically creating the widget command	++	++	++	++	+	+
Subcommands						
defining new subcommands	++	++	++	+	+	+
automatically parsing subcommands	++	++	++	○	+	+
fallback behaviour	+	+	+	+	+	?
Configuration options						
defining new configurable options	+	+	+	+	+	○
defining option handlers	+	+	+	+	+	○
add option handlers without parsing	+	+	+	○	○	○
automatically parsing options	+	+	+	○	○	?
manual propagation of options	○	○	○	○	○	○
automatic propagation of options	○	○	+	○	○	○
renaming options when propagating	○	○	+	○	○	○
Access to component widgets						
abstract names for components	○	○	++	○	○	○
hiding some abstract names from user	+	+	?	○	○	○
providing procedure to return path name	○	○	++	○	○	○
Reuse of widgets						
Reuse existing Tk widgets	○	+	+	+	—	?
Inheriting subcommands	+	+	++	+	○	—
redefining subcommands	+	+	++	○	○	—
access to original	+	+	+	○	○	—
Inheriting configuration options	+	+	+	+	○	—
redefining configuration handlers	+	+	+	○	○	—
access to original	+	+	+	○	○	—
Reuse with any encompassing widget	○	○	○	—	—	○
Miscellaneous						
Top-level widget support	○	○	○	+	+	○
Namespace support	++	++	++	—	+	—
Automatic database handling	○	○	+	+	+	—
Standard installation	++	++	++	++	+	+

++	supports very well
+	supports
○	doesn't support but possible
—	doesn't support
?	don't know

Table 3: Widget Builder View

4 Discussion

The previous evaluation tables guide this discussion on the various extensions. While our criteria are not the only ones that may be used, our experience with mega-widgets indicates that having these features in an extension makes programming easier for both the application and the mega-widget builders.

4.1 Application Builder View

The application builder's view is fairly well supported by most of the extensions. For instance, all of the extensions support standard Tk widget behaviour well since they all have a widget creation command, widget command, configurable options and widget subcommands. However, keep in mind that a "S" in the table only means that it is *possible* to create widgets satisfying this aspect of the application builder's view and it may depend on what the widget builder provides. For example, many extensions were given a "S" for "access to component widget" because it was possible, but only [incr Tk] defines a standard mechanism (the `component` subcommand) for doing so.

A common area where all of the extensions are questionable is Tk command support for mega-widgets and proper bindings for mega-widgets. For example, the `bind` command does not scale to mega-widgets; it is not clear which component widget (if any) should receive an event. Similarly, `focus` returns one of the component widgets rather then the mega-widget itself. Another command that poses a problem is `winfo`. Specifically, `winfo children` returns the components of the mega-widget instead of an empty string.

4.2 Widget Builder View

In contrast to the application builder's view, there is very little support for the widget builder. A number of the extensions come close to providing all that is needed, but some really miss the mark. One issue that arises is the tradeoff between flexibility and ease of use. For instance, the extensions that automatically create the encompassing frame do not provide a mechanism to *not* do this. The only extension that manages to retain flexibility while still automatically doing a large portion of the widget builder's work is [incr Tk].

4.2.1 Supporting the Application Builder View

Almost none of the extensions help the widget builder with the various components required for supporting the application builder's view. The extension that does it best is [incr Tk]. All of the extensions create the widget creation command, have support for subcommand implementation, and have the basic support for implementing configurable options. All of the extensions (except [incr Tk]) need to add support for the various methods of propagating options. This is a very useful feature, allowing the widget builder to list what is to be propagated to a component widget instead of having to define procedures to handle this. The extensions also need to provide a better means for accessing the component widgets. Four of the extensions have used common methods from object-oriented programming to hide options by allowing the widget builder to declare options as public or private.

4.2.2 Reuse of the Widgets

There are two different methodologies used to meet the requirements for reuse. [incr Tcl]-based systems achieve reuse by using class-based inheritance which handles subcommand reuse very well, but by itself does not handle the configurable options properly. The second method, used by TkMegaWidget, is an instance-based customization which only handles reuse moderately well. The current trend in the extensions appears to be a class-based inheritance method. However, it still remains an open issue as to which is better.

4.2.3 Miscellaneous

In order to maintain consistency with Tk, top-level widget creation is a necessity and should be a part of the extensions. Automatic option database handling and namespace support are not necessary but very useful in mega-widget applications, although they have their own set of problems. For instance, namespaces avoid name clashing but exhibit similar problem with `bind` not unlike those experienced with mega-widgets. Finally, extensions must install in standard ways, as those that do prevent many hours of frustration!

5 Conclusion

This paper has developed a framework for evaluating mega-widget extensions to Tcl/Tk. The framework was divided into two parts: the needs of the mega-widget user and the needs of the mega-widget builder. This evaluation framework was then used to evaluate six Tcl/Tk extensions.

From the evaluation we have identified four important issues for mega-widget extensions. First, the tradeoff between ease of use and flexibility; extensions that are easy to use often restrict their flexibility. Second, many of the "housekeeping" chores such as automatic parsing are very useful to provide, but care must be taken to maintain flexibility. Third, some of the Tk commands, such as `focus` and `winfo`, need to be extended in order to support mega-widgets. Fourth, there is the open question regarding reuse: whether class-based inheritance or instance-based customization is better.

6 Acknowledgements

I would like to thank Mark Roseman for all his patience and guidance, Saul Greenberg for providing me the opportunity to do this work, and John Aycock for his proof reading and technical advice. Ioi Lam, Steve Uhler and Greg McFarlane provided valuable commentary.

7 References

[1] S. Delmas, "Writing Tk Widgets with the MegaWidget." (included in distribution of TkMegaWidget)

[2] G. Howlett, "Packages: Adding Namespaces to Tcl," *Proceedings of Tcl/Tk Workshop*, 1994.

[3] M. J. McLennan, " [incr Tcl]: Object-Oriented Programming in Tcl," *Proceedings of the Tcl/Tk Workshop*, University of California at Berkeley, June 10-11, 1994.

[4] M. J. McLennan, " [incr Tk]: Building Extensible Widgets with [incr Tcl]," *Proceedings of the Tcl/Tk Workshop*, 1994.

[5] J. Ousterhout, *Tcl and the Tk toolkit*, Addison-Wesley, 1994.