In Search of a Customizable and Uniform User Interface

Bradley M. Kuhn*
bkuhn@cs.loyola.edu
http://www.loyola.edu/homes/bkuhn

David W. Binkley
binkley@cs.loyola.edu

4501 North Charles Street
Baltimore, MD 21210-2699
410-617-4354

1 Introduction

Modern software development techniques lack effective methods for building good user interfaces. Today's *ad hoc* methods require a significant proportion of development time, while the resulting interfaces tend to fall short in key areas. Previous attempts to ease the burden of designing and building good user interfaces also fall short of expectations.

A *good* user interface is both customizable and uniform. A customizable interface allows a user to change the manner in which she or he interacts with the application. A uniform interface provides the user with the same look-and-feel. Uniformity applies to interactions within a given application and to interactions throughout a set of applications.

The main component of a user interface is its *command language* (the text commands, button clicks, etc. that the application accepts from its user). For example, `ftp` (file transfer protocol) supports the commands listed in Table 1. `Ftp`'s command language was not obtained from an embeddable command language, rather it was written exclusively for `ftp`. This has three implications. First, `ftp` does not share a uniform interface with other similar applications. Second, building a uniform interface would have increased `ftp`'s development time and cost. Third, including customizability in the command language would further increase development cost.

A new approach to providing good user interfaces is the use of *embeddable command languages* [1]. When incorporated into an application an embeddable command language becomes the application's command language. Embeddable command languages differ from traditional command languages (such as `ftp`'s) in

*supported in part by a Loyola College Hauber Fellowship

that they are *application independent*. This makes adding a powerful command language to an application easier for the developer. An application inherits *for free* a customizable and uniform interface from the embeddable command language. This has two advantages. First, since an embeddable command language is written once rather than anew for each application, more resources can be devoted to producing a highly customizable interface. Second, if the same embeddable command language is incorporated into a collection of applications then these applications will share common instructions that come from the embeddable command language (*e.g.*, ringing the bell, setting colors, and looping structures). Thus, they have uniform interfaces.

However, embeddable command languages are not the panacea for all user interface problems. They require a sophisticated user to write the command language programs necessary to realize the customizability of the resulting interface. In this article, we describe *embeddable interfaces*: an extension of embeddable command languages that, when embedded into an application, provide a customizable and uniform interface for sophisticated as well as unsophisticated users. An embeddable interface increases the versatility of an embeddable command language by providing an interface to the command language that does not require users to write command language programs. However, more sophisticated users can still write such programs.

2 Background

This section first defines some vocabulary terms. It then provides details on customizability and uniformity, background on command languages, embeddable command languages, and Tcl.

2.1 Terminology

When discussing *embeddable interfaces*, it is important to clarify some related concepts and ideas. For example, programmers can be both writers and users of programs. This can cause confusion when an application includes a command language that allows the writing of programs, procedures, functions, or scripts that extend the application's interface.

Command	**Description**
`get`	download a file from a remote site
`send`	upload a file to a remote site
`binary`	switch to binary transfer mode

Table 1: Common commands in `ftp`'s command language

There are also the concepts of *embeddable* and *embedded.* Although The words are very similar, their meaning reflects different stages of a command language's use.

We use the following the following terminology in the remainder of this article:

Application	A program written by a developer to be used by the different levels of users introduced in Section 3.
Developer	A programmer that writes applications; not the user of these applications.
Program	Code written by a programmer to extend an application's user interface.
Programmer	A user who can write code.
User	A user who may or may not write code.
Command Language	A language that the user uses to interface with an application (further discussed in Section 2.3).
Embeddable Command Language (ECL)	An application independent command language that can be placed in any application (further discussed in Section 2.4).
Embedded Command Language	an ECL after it has been embedded in an application

2.2 Customizability and Uniformity

Customizability refers to the number of things that a user has control over. Customizability is an important factor in the usefulness of an application, since a highly customizable application allows its user to "teach" the application the manner in which the user wants to use it. Such applications better meet users needs by not forcing the user to interact "its way."

A uniform interface conforms to some look-and-feel standard. Uniformity allows experience with one application to be applied to a new application. For example, a user who is accustom to entering 'q' to quit an applications, would like to enter 'q' to quit a new application rather than, for example, 'e' to exit the new application. New applications are easier to learn if they share common commands with existing applications. The same is true for learning a new feature of an existing application. If the commands of the new feature are the same as the commands of a previously known feature, the new feature will be easier to learn.

Industry development environments for building user interfaces (*e.g.*, widget sets, GUI standards) tackle only the uniformity problem. Applications written in these environments share a common look-and-feel. However, the user has little chance to customize the interface. For example, applications written using the Motif [1] environment have a uniform interface. However, there is little support for customization in these environments. A user, for example, cannot make all Motif applications have a function key that moves the "scroll-bar" to the top of the window. An interface built from an embeddable interface would easily provide such options.

[1] Motif® is a registered trademark of the Open Software Foundation, Inc.

WordPerfect	**Lotus 1-2-3**
`{BELL}`	`{beep}`
`{PROMPT}Paging Down Now~`	`{indicate Paging Down Now}`
`{Page Down}`	`{pgdn}`

Table 2: A comparison of WordPerfect's and Lotus 1-2-3's command languages

2.3 Command Languages

Most applications have a command language. Most provide little customizability, however some were designed with customizability in mind, and thus are quite flexible in this regard. For example, many people are familiar with the command language of GNU Emacs [4]. GNU Emacs' command language is based on Lisp, and it allows the user to issue commands that bind certain keys to specific actions. The user can "teach" Emacs respond to certain keys in new ways. In addition, command scripts can be written to perform additional actions not initially provided by Emacs.

Two costs of customizability in a user interface like Emacs' are the cost to the developer in building this functionality and the cost to the user in understanding the interface. Building good command languages is a difficult and time consuming task. The cost to the user is high because different applications often have different command languages. The user must learn each to customize each application. This is made worse because these languages are often proprietary. Thus, other developers find it easier to design a new (and different) command language than to deal with another organization's proprietary command language.

An example of this "lack of compatibility" between user interfaces is shown in Table 2, which compares the command languages of WordPerfect[2] and Lotus 1-2-3.[3] Each column contains the commands necessary to sound the terminal bell, inform the user that a page down will occur, and then perform the page down. As is clear from Table 2, the languages have little semantic difference, but great syntactic difference. This slows learning one from the other.

2.4 Embeddable Command Languages

ECLs were proposed by John Ousterhout [1] as an alternative to traditional command language development. An ECL is an application independent command language that is made part of (embedded into) an application. Once embedded, it becomes the command language for the application. Figure 1 diagrams how an ECL fits into an application.

To understand the power of ECLs, consider two existing batch applications: one that can calculate the

[2] WordPerfect® is a registered trademark of the WordPerfect Corporation.
[3] Lotus® and 1-2-3® are registered trademarks of Lotus Development Corporation.

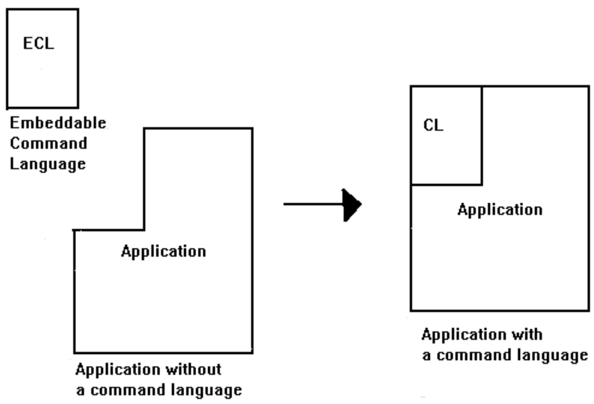

Figure 1: How an ECL fits into an application

```
cl> set equation = "x^2 + 2*x - 5"
cl> answer = derivative(equation)
cl> print(answer)
  2*x + 2
```

Figure 2: An example of a derivative program embedded with a command language

derivative of an equation, and one that provides statistical functions such as `findMean`. To provide an interactive derivative calculator, a developer could simply write a bridge function that permitted an ECL to access the derivative program. When the user ran the resulting application, the user would interact with the resulting command language. This is illustrated in Figure 2, where the user types the `set` command, setting the variable `equation` to the given value, uses that variable in the call to `derivative`, and finally `prints` the resulting answer. The developer only wrote a bridge function from the ECL to the derivative calculator. The command language provides the interactive interpreter, including the `set` and `print` commands.

The interactive derivative calculator provides two kinds of commands: commands that calculate derivatives, and commands that set variables, execute loops, and call procedures. If the statistical application uses the same ECL, the resulting application would again provide two kinds of commands: commands that do statistical calculations (such as `findMean`), and the **same** commands as the derivative calculator that set variables, execute loops, and call procedures.

Two advantages of ECLs become apparent. First, the simplicity of creating a good user interface: a developer simply writes a bridge from his application to the command language and his application automatically inherits a good user interface. Second, the user need only customize one of these applications and the same customization may be transferred to others. For example, a user may want to replace the `set` command with a `let` command. After this was done for the derivative calculator, the same customization is easily applied to the statistical application.

2.5 Tcl—An Embeddable Command Language

Tcl (Tool Command Language) is the most popular readily available ECL [1]. It is a small interpreted ECL with a large number of basic commands that can be extended by the developer. Tcl also has an interface to C, so that it can be embedded into C applications.

One application that has helped to popularize Tcl is Tk, a X window system toolkit written using Tcl and C [2]. Tk provides a unique outlook on ECLs. It is both an ECL and an application that was developed with an ECL. When using Tk, a developer is, in one sense, simply a user of a Tcl command language. The developer uses the command language that Tcl provides and the X windows extensions that Tk provides. Thus, the Tk "developer" is also a Tcl "programmer."

3 User Capability Levels

A major difficulty in producing a customizable and uniform user interface is the existence of different user capability levels. This section considers the four levels shown in Table 3. Each level has its own traits that dictates how its users interact with programs.[4] These levels are based on those proposed by Ben Schneiderman [3].

The first level contains *beginning users*. These users want to push a button and have the correct action take place. They often have a difficult time understanding complicated user interfaces. This is unfortunate,

[4]Note that a user who could fit into one category often chooses to be in another category (*e.g.*, an experienced programmer may prefer to be an end user when running some applications).

User Levels	**Attitudes about Customization**
beginning user	cares little about customization
experienced user	knows how to customize by setting options
inexperienced programmer	writes simple scripts and procedures
experienced programmer	writes large amounts of code

Table 3: Levels of users with respect to customization

since these users stand to benefit the most from customizable and uniform interfaces (*i.e.,* the less sophisticated the user, the more assistance he or she requires from an application). Making the power of ECLs accessible to less sophisticated users would increase the users' productivity.

The second level contains *experienced users* who are not programmers. Experienced users typically have figured out how to edit customization files in order to get an application to behave in a manner that best suits their needs. These users do not want to write full-fledged programs in a command language, but they may write "one-liners" and set values of variables to make an application behave closer to their needs.

The third level contains *inexperienced programmers* who are capable of writing simple programs in order to make an application better suit their needs. Users who have written simple UNIX[5] shell scripts are at this level.

The fourth level contains *experienced programmers.* These users are the ones who can utilize the full capability of a command language. These users get the most out of a program that provides customization through code writing.

Current applications are usually targeted toward one user-capability level. If users of different levels are to use an application, the developers usually have to target for the lowest level. For example, an application that users of all levels will use would have to be designed for the beginning user level. This frustrates more advanced users.

4 Current User Interfaces

This section examines current user interfaces that attempt to provide customizability and uniformity. It first considers custom (not customizable) user interfaces. Next, it considers applications that provide a command language. Then it considers the interfaces that result when Tcl is embedded into an application. None of these provide customizability and uniformity to all four levels of users. The next section introduces *embeddable interfaces*, which provide customizability and uniformity to all four users levels.

Currently, there are few applications that attempt to provide the functionality necessary to allow users of different levels access to customizability. One existing compromise is seen in the X windows resource database, which allows some user flexibility: applications can have end user level interfaces, which modify the database, while experienced users can set options directly by modifying a *resource database file.* This model partially accommodates end users and experienced users. UNIX shells provide a similar crossover between inexperienced and experienced programmers. Inexperienced programmers can use the shell to write one line pipelines and other useful commands. Experienced programmers can write complex shell scripts.

These examples illustrate limitations in present user interfaces. For example, experienced programmers cannot completely customize X windows applications; UNIX shells have a long history of baffling beginning

[5]UNIX® is a registered trademark of AT&T Bell Laboratories, Inc.

users. We call these interfaces *custom user interfaces* because they are *customized* to one or two user levels, but do not span all user levels.

In contrast, programs like GNU Emacs have reached a plateau of spanning all user levels. Simple end users can set options via interactive editor commands. Experienced users can create an initialization file that initializes some customization settings. Inexperienced programmers can write short macros and scripts that perform repetitive editing functions. Finally, experienced programmers can easily rewrite and replace vital parts of Emacs with custom code.

Emacs seems to be the solution that we seek. However, developers spent a considerable amount of time building the interface. Furthermore, this work is specific to Emacs, and cannot be easily reused. Tcl is an attempt to lessen the difficulty in developing a command language similar to Emacs' command language. However, Tcl does **not** provide a "ready-made" interface that spans all user levels. Gaininger the full use of a Tcl embedded command language requires knowing or learning a fair amount of the syntax and semantics of Tcl. Thus, applications that embed Tcl are often targeted toward programming users. This seems to contradict the original intent of Tcl, which was to provide simple user interfaces. Unfortunately, the less programming a user is willing to do in an embedded command language, the less customizable the application becomes. Note that even experienced programmers who could learn an embedded command language may not *want* to learn it. In the extreme, the ECL becomes a sophisticated tool known only to the developer. It would be beneficial if the power of an embedded command language could be extended to encompass different levels of users.

5 Embeddable Interfaces

When an embeddable interface is embedded into an application, it provides an interface that is customizable and uniform to all four user levels. Embeddable interfaces are an extension of ECLs to allow all user levels to interact with the application in the manner with which they are most comfortable.

The most intriguing aspect of embeddable interfaces is that they are embeddable, just like an ECL. Developers incorporate embeddable interfaces into their own application by providing a list of configurable options in their application. The embeddable interface then provides configuration windows and features that allow different levels of users to configure the application in different ways. In this manner, low-end users have a customizable and uniform interface and high-end users have a customizable and uniform command language interface.

5.1 Implementation

Our embeddable interface has been implemented using Tcl, and obtains common look-and-feel through its use of Tk. The interface has four layers that correspond to the four user levels. Each layer provides the user with an appropriate interface. The multiple layers integrate with each other, so that users can overlap multiple levels.

The end users' layer of the interface is by far the most difficult to implement since its functionality is the farthest from that provided by Tcl. This layer provides a simple windowed interface for customizing various options of an application. All configurable options are stored in a list that the embeddable interface uses to determine how to organize the information on the screen. The windowed interface is automatically generated from the application by scanning a structured list and building a window with various, easy-to-use widgets that allow the end user to configure the application. An example window, taken from the application discussed in Section 5.2, is shown in Figure 3. Figure 4 shows the *complete* code that the developer provided to created the structured list for the embeddable interface.

The experienced user layer provides the ability to edit options in a text format similar to the X resource database. This more concise representation allows faster and easier editing, but requires greater sophistication on the part of the user.

The beginning programmer layer includes a "hook" mechanism similar to the one that exists in Emacs. This allows the user to write short scripts in the embedded command language that become part of the embeddable interface. These scripts are executed before or after a major event occurs in the application (*e.g.*, the opening of a new window). The user can set parameters and otherwise modify the behavior of an application. While these customizations could be a achieved at either of the lower levels, having them under program control allows, for example, different options to be selected on Tuesdays than on Fridays.

The experienced programmer layer provides direct access to the embedded command language. However, an extension is made. Because of Tcl's use of global variables, a programmer who uses a variable name that is the same as one reserved for the embeddable interface, unpredictable behavior may result. (The same is true for beginning programmers as well.) The protection mechanism prevents programmers from accessing or changing internal data that would cause problems for the application or its interface.

5.2 Example Application

In addition to developing the embeddable interface itself, we also implemented an application that embedded our interface. The application is an X windows *session manager* (similar to DEC's[6] `dxsession`), which manages information about user sessions (logins). This includes managing multiple windows, mouse characteristics, etc.

For the beginning user level, windows such as the one shown in Figure 3 are provided. The experienced user is provided with a configuration file that could be edited to configure these same options. The beginning programmer is given the options to add functions that are executed at key points in the startup and shutdown of a session. The experienced programmer can actually edit parts of the code that are not protected for internal use only.

[6]DEC® is a registered trademark of Digital Equipment Corporation.

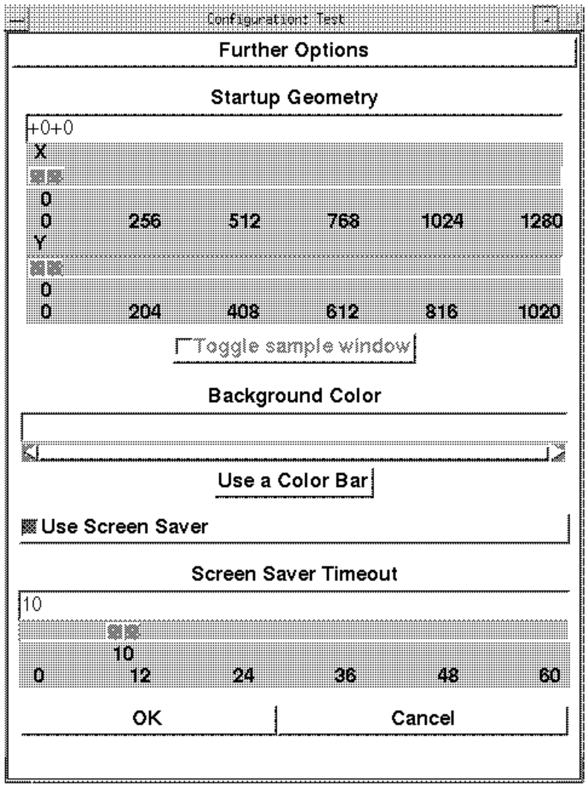

Figure 3: A sample customization window from the first layer

```
proc Tkbd:Test:DefaultInit { args } {
global Test

keylset Test(options)
  furtherOptions.optDescription              “Attributes of the window”
  furtherOptions.minimumX.optType            scale
  furtherOptions.minimumX.optValue           10
  furtherOptions.minimumX.optChoices         [list 0 $max(x) $interval(x) $length(x)]
  furtherOptions.minimumX.optDescription     “Minimum x pixel value of the window”

  startupGeometry.optType                    geometry
  startupGeometry.optValue                   “+0+0”
  startupGeometry.optChoices                 [list 0 $max(x) $interval(x) $length(x)
                                             0 $max(y) $interval(y) $length(y)]
  startupGeometry.optDescription             “Geometry setting used to startup the session manager’

  backgroundColor.optType                    color
  backgroundColor.optValue                   “”
  backgroundColor.optDescription             “Default window background color”

  useScreenSaver.optType                     checkbox
  useScreenSaver.optValue                    1
  useScreenSaver.optDescription              “Enable/Disable screen saver feature’

  screenSaverTimeout.optType                 scale
  screenSaverTimeout.optValue                10
  screenSaverTimeout.optChoices              [list 0 60 12 1]
  screenSaverTimeout.optDescription          “Amount of time before the screen saver engages’

Tkbd:Init:ValidateArray Test

}
```

Figure 4: Code to build the list structure for the first layer

It is important to note that the embeddable interface also makes this application very customizable at all four levels. However, very little code was required to add this customizability. Our embeddable interface provides much of the customizability simply by writing bridge functions between the application and the embeddable interface.

This example application developed with the embeddable interface shows that both ECLs and embeddable interfaces make application development easier, and make it more cost effective to bring customizable and uniform interfaces to all levels of users.

Acknowledgment

Dennis Smith helped in the design and implementation of the session manager application.

Michael Kuhn originally suggested that ECLs might be a useful area to research.

Figure 1 was originally drawn by Julie Pointek.

References

[1] John K. Ousterhout. "Tcl: An Embeddable Command Language". In *Proc. USENIX Winter Conference*, January 1990.

[2] John K. Ousterhout. "An X11 Toolkit Based on the Tcl Language". In *Proc. USENIX Winter Conference*, January 1991.

[3] Ben Shneiderman. *Designing the User Interface.* Addison-Wesley Publishing Company, Reading, Massachusetts, second edition, 1992.

[4] Richard M. Stallman. *GNU Emacs Manual.* Free Software Foundation, Cambridge, eight edition, June 1993.