Actions: A Simple Implementation

Bryan Oakley
bryan@bitmover.com

October, 2004

Introduction

Writing rich, highly interactive event driven user interfaces offer challenges not present in other types of programs. I define highly interactive programs as those in which the controls constantly reflect the state of the application. For example, a cut button or menu item would be disabled in the absence of a selected object or span of text.

Rich user interfaces present the user with multiple ways to accomplish a task. For example, again considering a traditional "cut" function, the application may provide a toolbar button, an item under the Edit menu of the menubar, a keyboard accelerator, and an item on a context sensitive (right-click) menu. A highly accessible application may even provide for speech input, mouse gestures or other input devices.

The Difficulty in Writing Rich Applications

Two large problems exist for developers of such user interfaces. One is the need to constantly synchronize the controls with the ever-changing state of the application or data. When no text is selected, for example, the cut and copy buttons should be disabled.

Another problem is that as programs evolve over time it can become tedious and error prone to update the parts of the code that act upon these controls. A traditional solution is to simply call the configure configure subcommand of each widget in order to set the state; this can become cumbersome as the number of widgets grows, or as the number of places that enable or disable the widgets grows.

Solving The Problem With Actions

One technique which can be used to solve both of these problems is to define end user functionality as pseudo-objects referred to in this paper as *actions*. Actions are much like procs in that they are defined with arguments and a body of code to be executed.

Unlike procs however, actions may be in an enabled or disabled state. Disabled actions will do nothing when invoked. Also, actions are bound to widgets or parts of a widget (in the case of menu items) so that any time the state of the action changes, the associated widget(s) are changed accordingly.

A Short Example

The following example defines an action named "copy" which arranges for the focus window to perform a copy operation. Two widgets and an accelerator are then associated with this action:

```
action define copy {} {event generate [focus] <<Copy>>}
button .toolbar.copy \
    -text "Copy" \
    -command [list action invoke copy]
.menubar.editMenu add command \
    -label "Copy" \
    -command [list action invoke copy]
action associate copy .toolbar.copy
action associate copy .menubar.editMenu "Copy"
bind all <Control-c> [list action invoke copy]
```

Initially the state of the copy action may be disabled since the user hasn't yet selected anything in the application. This would be accomplished with the following code as part of the start up sequence:

```
action disable copy
```

When the action is disabled, any widgets that are associated with the action will automatically be disabled. Later the application may detect that the user has selected something, in which case it would enable the copy action and thus enable all associated widgets:

```
action enable copy
```

What happens if later a requirement comes down to add a context sensitive menu or some other method of invoking the function? All the programmer must do is create the new widgets and associate them with the action. The widgets will then automatically be enabled or disabled at the appropriate times without having to make any other modifications to the application:

```
menu .popup
.popup add command \
    -label "Copy" \
    -command [list action invoke copy]
```

Additional Benefits of Actions

When designing applications around the concept of actions, a good rule of thumb is to create an action for every discrete bit of work the user can perform. Cut would be an action, copy would be an action, paste would be an action, and so on.

By sticking to this rule, actions can provide additional benefits. For example, if all user interaction is via actions it would be possible to modify the action code to log all user interactions in order to determine which features are used most often by your customers. Or, actions could be saved in a stack to implement a high level undo, or they could be combined to create a macro facility.

Actions can also be used to help automate the process of gui testing. For example, one might write a test script like this:

```
action invoke insert 1.0 "Hello, world\n"
action invoke select 1.0 1.5
action invoke copy
set clipboard [clipboard get]
if {$clipboard ne "Hello"} {
    puts stderr "test failed"
    exit 1
}
```

Actions could also be used to provide an API for third party developers. With actions themselves defined through a common API it becomes simple to automatically generate API documentation.

A Simple API

There are a myriad of ways to introduce the concept of actions into an application. This paper presents a very simple API that performs the basic functions of defining actions, associating them with widgets, and enabling and disabling the actions:

action define *actionName args body*

This behaves much like the proc command, defining a body of code to be executed when the action is invoked.

action associate *actionName pathName ?index?*

This associates an action with a widget or menu item. When an action is enabled or disabled, so will the widget or menu item be enabled or disabled.

action enable *actionName ?actionName ...?*
action disable *actionName ?actionName ...?*
action state *actionName*

These three commands set or query the state of an action. Actions are always either in a "normal" or "disabled" state.

action invoke ?-force? *actionName ?args?*

Unlike the proc command which creates a new command by a given name, this implementation requires that actions be invoked via "action invoke". If the action is disabled, invoking it will have no effect unless the -force option is given.

The full implementation, which is only about 150 lines of code, appears at the end of this paper.

Alternate Implementations

This paper presents one simple API for implementing actions, but there are many different ways to implement the same concept. For example, actions could be implemented as SNIT objects with methods for creating associated widgets, e.g.:

```
action copyAction {} {event generate [focus] <<Copy>>}
...
copyAction button .toolbar.copy -text Copy ...
copyAction menu .editMenu add command -label Copy ...
...
copyAction enable
```

Another alternative is to associate additional widget attributes with an action in addition to state so that all widgets and menu items have identical labels or images:

```
action create copy \
    -text "Copy" \
    -image copyImage \
    -command
button .toobar.copy
action associate copy .toolbar.copy
```

Conclusions

Actions have many uses and many benefits, with negligible costs. No matter what form the API takes, the ability to associate a set of widgets with one another, and to be able to enable or disable all widgets that perform a specific function with a single command, is a powerful technique for creating highly interactive, rich user interfaces.

While it is certainly possible to develop rich, highly interactive applications without using actions, it is my experience that using actions leads to a cleaner internal design which in turn results in a higher quality application with lower maintenance costs.

Actions.tcl

```
# actions.tcl
#
# usage:
#
# action define actionName args body
# action associate actionName pathName ?index?
# action enable actionName ?actionName ...?
# action disable actionName ?actionName ...?
# action state actionName
# action invoke ?-force? actionName ?args?
#
package provide actions 1.0

namespace eval ::action {
    variable action
    variable actionState
    namespace export action
}

proc ::action::action {subcommand args} {
    variable action
    variable actionState

    set result ""
    switch -exact -- $subcommand {
        "enable" -
        "disable" {
            if {[string equal $subcommand "enable"]} {
                set state normal
            } else {
                set state disabled
            }
            foreach actionName $args {
                if {![info exists actionState($actionName)]} {
                    return -code error "action $subcommand:\
                                    unknown action \"$actionName\""
                }
            }
            foreach actionName $args {
                ::action::setState $actionName $state
            }
        }

        "associate" {
            set actionName [lindex $args 0]
            set args [lrange $args 1 end]

            if {![info exists actionState($actionName)]} {
                return -code error \
                    "action associate: unknown action \"$actionName\""
            }
            if {![info exists action($actionName)]} {
```

```
            set action($actionName) [list $args]
        } else {
            lappend action($actionName) $args
        }

        ::action::setState $actionName

        # return a result that could be used as an argument
        # to a -command option of a widget.
        set result [list action invoke $actionName]

    }

    "state" {
        set actionName [lindex $args 0]
        if {[info exists actionState($actionName)]} {
            set result $actionState($actionName)
        }

    }

    "define" {
        set actionName [lindex $args 0]
        set actionArgs [lindex $args 1]
        set actionBody [lindex $args 2]
        proc ::action::action-$actionName $actionArgs $actionBody

        if {![info exists actionState($actionName)]} {
            set actionState($actionName) "normal"
        }
        set action($actionName) [list]
    }

    "invoke" {
        # normally, disabled actions won't do anything. If
        # the first argument after invoke is "-force", it will
        # run the action regardless of the state. This is useful
        # in scripts. The default behavior of not doing anything
        # if the action is disabled is quite handy when calling
        # actions from keybindings.
        if {[lindex $args 0] eq "-force"} {
            set actionName [lindex $args 1]
            set args [lrange $args 2 end]
            set ok 1
        } else {
            set actionName [lindex $args 0]
            set args [lrange $args 1 end]
            if {[info exists actionState($actionName)] &&
                $actionState($actionName) != "normal"} {
                set ok 0
            } else {
                set ok 1
            }
        }
```

```
            if {$ok} {
                set command [linsert $args 0 \
                                 ::action::action-$actionName]
                #                       set result [eval $command]
                set result [uplevel $command]
            } else {
                set result ""
            }
        }
    }

    return $result
}

# called with a null state, will re-configure all associated
# widgets to be in sync with the current state. Useful when associating
# new widgets with an action
proc ::action::setState {actionName {newState {}}} {
    variable action
    variable actionState

    if {[info exists action($actionName)]} {
        foreach association $action($actionName) {
            if {[string equal $newState ""]} {
                set state $actionState($actionName)
            } else {
                set state $newState
            }
            set widget [lindex $association 0]
            if {[string equal [winfo class $widget] "Menu"]} {
                set index [lindex $association 1]
                $widget entryconfigure $index -state $state
            } else {
                $widget configure -state $state
            }
        }
    }
    if {![string equal $newState ""]} {
        set actionState($actionName) $newState
    }
}
```