Maintainable, Shareable and Easily Creatable & Updateable toolbar, menubar, statusbar - pillars of any GUI application.

Tarun Goyal

Abstract – This paper presents a novel approach to efficiently manage, update and share the *toolbar, menubar & statusbar* widgets that are integral to any TCL/TK based GUI application. However, considering that any GUI would have different windows performing variety of tasks and be dependent on the overall tool state, the solution should effectively support context sensitivity with respect to windows, selected object in its constituent windows and tool status.

Fig: Typical Menubar/Toolbar in a GUI

Summary - It has been observed that in any GUI application the onus of creating/updating the widgets inside the toolbar, menubar or statusbar lies with the developer responsible for creating a component [or window/frame] in the GUI with everyone creating their own "versions", resulting in code duplication and raising maintainability issues. However, considering that same widget [e.g. button, menus etc.] might perform similar function in various windows and/or the same real estate could be utilized to create different widgets for different windows, necessitates a functional requirement to have centralized *toolbar, menubar & statusbar* managers [or *smart widget managers*] that helps one to easily register widgets and update them dynamically based on the current window [or the element therein] in focus. Essentially, following are the desired features of these centralized "managers".

1. Each manager should be structured in a way so as to provide a centralized mechanism for creating/updating [e.g. enabling/disabling/setting a value] the widgets.
2. Ability to create a variety of widgets – e.g. a *checkbutton* or *menu* in a *menubutton*, *text-widget* or *button* in a *toolbar* or *progress-bar* or a *labelframe* in a *status-bar*.
3. Make the same widgets as reusable as possible – e.g. a "cut" button can be used to cut a text item in one window whereas could be used to cut a schematic element in another.

The following sections captures the pseudo implementation interface [written in *incrTcl*] of various managers, the central repositories that will entertain requests received from any window/component. Please note that only "major" interface functions have been mentioned here and the managers may contain some other methods from implementation perspective. For the complete implementation, please refer the supplied TCL source code.

TOOLBAR MANAGER

```
itcl::class ToolBarManager {
        #variable list
        set dock_bars( std, tree, browser, schematic, dataview)   ## dockbars of tool-bar
        set dock_bar_widgets (copy, cut, paste, find ..... )      ## widgets in the dock-bar

        ## widget_prop store the properties of the widget  - e.g. label, underlying index etc.
        set widget_prop[std, copy] = { icon_name, default_callback}
        ## default_callback is the method to call on  "invoking" this widget
```

```
        # Tool Bar associated with a window:
        set windowToolBar(windowName) = {toolBarObject}

        # this will store the global tool-bar object of the main framework
        private  variable  globalToolBarObject

        # Register_Window::  This  function enables a window to register associated docbar and
        # the widget
        private method  Register_Window  { windowName, docName, widgetName }

        # Unregister_Window:: This function deregisters the given window from relevant dockbar
        # lists
        private method  Unregister_Window {windowName }

        # Update_ToolBar :: This proc is the updates the respective toolbar based on the window
        private  method Update_ToolBar  { windowName }

        ## Update_DocBar :: This function updates the requested dock-bar, with the latest-status
        ## of the widgets residing inside the dock-bar.
        private method  Update_DocBar { docBar_Name }

        # Create_DocBar :: Create a dock-bar with a given doc-bar-name..
        private method  Create_DocBar { docBarName }

        #  Create_Widget :: Create a widget in the specified doc-bar, with the given widget-name
        private method  Create_Widget { docBarName  buttonName }

        # this will handle the call-backs, smartly finds the current window object and calls it
        # method.
        public method CallBackHandler { widgetName  docBarName}

        #  this  method  changes the state of buttons  depending  upon the current selection inside
        # a window
        public method updateState { windowName  docBarName widgetName }
    }
```

The Toolbar manager is instantiated in the constructor as follows-
set globalToolBarObject [mtiwidgets::dockbar $_vars(debug_win).$toolBarName -relief sunken -borderwidth 1]

Some of the salient features, among others, of the toolbar manager are as follows-

1. Only those *dockbars* [*dockbar* is a subunit of a "toolbar"] that are registered with current set of visible windows shall be shown and the widgets inside these *dockbars* are enabled/disabled as per the window requirements. Further the *dockbars* are added/deleted incrementally as windows are shown/hidden in the tool.

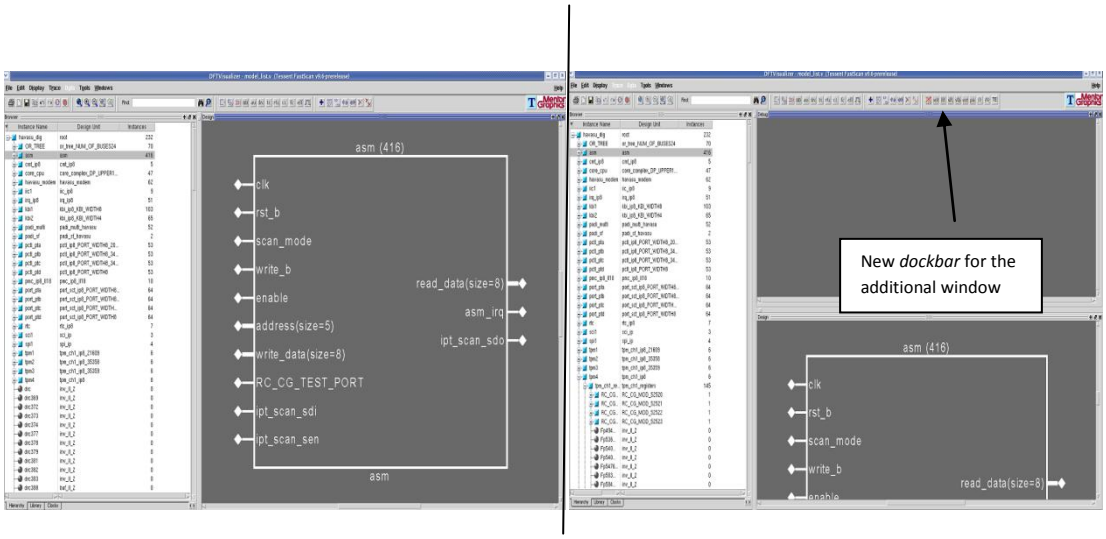

Fig: Tool in 2 different states on opening a new window

2. As per our present tool requirements, the currently supported widgets are – *buttons*, *entry*, *combobox*, *checkbutton*, *radiobutton*. The manager can easily be enhanced to support more widgets.
3. The manager also supports placing the window specific widgets in the undocked state i.e. in case user undocks [i.e. does a "*toplevel .$windowName*"] the window from the tool, only the widgets that are registered with that window comes in the undocked toplevel window. A separate instantiation happens for the toolbar inside the undocked window.
   set windowToolBar($windowName) [mtiwidgets::dockbar "[$frameworkHandle getPaneManager].$windowName.toolbar" -relief sunken -borderwidth 1]

MENUBAR MANAGER

Similarly, the Menubar manager looks as follows:

```
itcl::class MenuBarManager {
 # Following data-structures are maintained by Task-Manager internally storing the dock-button
 # related properties
 private variable globalMenuButtons, menuButtonMenus, menuProps, menuButtonProps

 # List of created menu-buttons are maintained as ::
 # createdMenuButtons {} = { std, ..... }
 private variable  createdMenuButtons {}

 # Array which tells us about the status of a menu-button ::
 # menuButtonStatus (std) = { "Enable" } .....
 private variable  menuButtonStatus

 # Global list of registered menu-buttons and menus ::
 # Reg_WindowMenuButton (docName) = {  windowName, .....}
 # Reg_WindowMenu (docName, ButtonName) = { windowName, .....}
 # Reg_WindMenuCallback(docName, ButtonName) = {  CallbackFunction , ..... }

 # this list will store the menus that you want to create dynamically
 # at each update. e.g. display->marking menu (format = [list "menubutton,menu" ..])
```

```
   private variable menuIsDynamic

   private variable  windowMenuBar ; # windowMenuBar(windowName) = {menuBarObject}
   # Register_Window:This  function enables a window to register associated doc-Name and
   # button-Name
   public method  Register_Window  { windowName menuButtonName menuName
callBackFunc }

   # Unregister_Window:: This function deregisters the given window from the
   # both the lists :: Reg_WindowMenuButton and Reg_WindowMenu.
   public method  Unregister_Window {windowName }

   # Update_MenuBar :: This proc  is called from a centric place and  depending on the
   # “windowName” updates the respective menu-bar. Showing/hiding the menu-bar is also
   # handled here.
   public method Update_MenuBar  { windowName Docked {forceTag "no"}}

   # Update_MenuButton :: This function updates the requested menu-button, with the  latest-
   # status of the menus residing inside the menu-button.
   private method  Update_MenuButton { menuButtonName object windowName
menuBarObjState {forceTag "no"}}

   # Create_MenuButton :: Create a menu-button with a given menuButtonName..
   private method  Create_MenuButton { menuButtonName menuBarObj Docked {enterTag 0} }

   # getUndockedMenuBarObj  :  This proc creates the object for undocked menu-bar with all the
   # contents that goes into the Undocked Window. This proc is called as soon as the window is
   # undocked.
   public method GetUndockedMenuBarObj { windowName  }

   # this will handle the call-backs
   public method CallBackHandler { menuName menuButtonName }

   # this  method  changes the state of menus  depending  upon the current selection inside a
   # window
   public method UpdateState { state windowName menuButtonName { menuName "" } }

   # this method handles tool-bars when a window is maximized
   public method maximizeWindowMenuBars { windowName }

   # method creates deleted menu-item
   private method Create_DeletedMenuItem { menuButtonName menuName menuObj \
insertIndex  Docked windowName {enterTag 0}}

  # function to test for the validity of menu-item in "data" menu-button for "design & debug"
   public method MenuItemValidForData { args }
```

```
  # this function sets the widget value to the instructed "value"
  public method setWidgetVal { windowName menuButtonName menuName value }

  # this function returns the value stored in widget
  public method getWidgetVal { windowName menuButtonName menuName }

  # this function sets the key index array
  public method setKeyIndex {menuNameList }

  # this function returns the key Index
  public method getKeyIndex { menuName }
}
```

The Menubar manager is instantiated in the constructor as follows-

set globalMenuBarManager [iwidgets::menubar $_vars(debug_win).$menuBarName –font $_fonts(helvB:12) -helpvariable helpstr]

Some of the salient features of the menubar manager are:

1. The menus are created/deleted on the fly and are dynamic in nature in that they can be created at run time. This is accomplished by various functions talking to each other in parallel - *Create_DeletedMenuItem* is called from *UpdateMenuButton* for each registered menu with the window with *Create_DeletedMenuItem* checking the validity of the menu by calling *MenuItemValidForData*. The "runtime" menus are stored in a special variable called *menuIsDynamic*, which is checked every time a menu-item is created.
2. Menubar manager supports widgets such as *menubutton*, *menuitem*, "*cascade*" [no limit], *checkbutton, radiobutton, sepators* and can be enhanced to support more widgets easily.
3. As with toolbar manager, menu-bar manager has also been implemented in a way so as to show the registered menubuttons with the undocked window in the toplevel window. Each window gets its own menubar manager object as follows-
   set windowMenuBar($windowName) [iwidgets::menubar "[$frameworkHandle getPaneManager].$windowName.menubar" -font $_fonts(helvB:12) -helpvariable helpstr]

STATUSBAR MANAGER

```
itcl::class StatusBarManager {
  # Following data-structures are maintained by Task-Manager internally
  private variable statusBarWidgets ; ## all the widgets in the status bar
  private variable widgetProps ; ## widget related properties


  # Global list of registered doc-bars and buttons ::
  # Reg_WindowWidgets = {  windowName, .....}
  # Reg_WindowWidgetCallback(widgetName) = {  CallbackFunction , ..... }
```

```
  private variable  Reg_WindowWidgets
  private variable  Reg_WindowWidgetCallback

  # public methods

  #  Methods that will be used by the tool windows
  # Register_Window::  This  function enables a window to register associated status bar widget
  public method  Register_Window  { windowName widgetName  {callBackFunc ""}}

  # this method return the value stored in the widget variable
  public method getWidgetVal { windowName widgetName }

  # this method sets the widget's variable value to "value", with "progressBarVal" is an
  # optional argument that will be valid in the case of "progressBar" widget.
  public method setWidgetVal { windowName widgetName value {progressBarText ""}}

  #  this  method  changes the state of widgets depending  upon
  # the current selection inside a window
  public method UpdateState { windowName widgetName State}

  #  Methods that will be used by framework
  # Update_StatusBar :: This proc  is called from a centric place and  depending on the
  # windowName updates the status-bar
  public method Update_StatusBar  { windowName Docked }

  # getUndockedstatusBarObj  :  This  proc will return object of the statusbar, that goes into the
  # Undocked Window.
  #This proc is called as soon as the window is undocked.
  public method GetUndockedStatusBarObj { windowName }

  # private methods
  # Create_StatusBar :: Create a status-bar with a given status-bar-name..
  private method Create_StatusBar { widgetName windowName }

  # this will handle the call-backs
  public method CallBackHandler { widgetName }

  # this function initializes the various status-bar related Lists
  private method initializeLists {}
}
```

The Statusbar manager is instantiated in the constructor as:

```
  set globalStatusBarObj [frame $_vars(debug_win)._bottomFrame]
```

As can be seen, statusbar is a frame widget and we are packing the widgets it.

Statusbar manager is similar to the above managers [mostly toolbar manager] and works in a similar fashion. Some salient features are:

1. Statusbar manager supports widgets such as – "*progressbar*", *entry, label*. We have created our own progress bar and have instantiated it inside the status bar. The function "*setWidgetVal*" handles the update process of the progress bar accepting %ages and/or text as an argument. Including progress bar has helped in creating significant value for our customers especially for operations that take time to complete.
2. As with other managers, *statusbar* manager created a separate instantiation for all the undocked windows and places the registered widgets inside that window. Each window has its own "status bar manager *frame*" in which the widgets are packed.

USAGE

A single object of each of the managers is instantiated inside the constructor of the respective class . This object could then be accessed and used by various windows to manage their *dockbar, menubar & statusbar* items, as captured in some ways below.

1. *Register a variable*: registering any widget with the toolbar [or statusbar] manager is easy. The client needs to call the following:
$toolBarManObject Register_Window $windowName $dockBarName $widgetName "CALLBACK FUNCTION"
Above has the syntax as – {Window/Component Name registered with widget, DockBar within the toolbar, Widget Name, method that will be called on "invoking" the widget}

In the toolbar manager, the following would be done for initializing it.
widgetProps($dockBarName,$widgetName) { "Print ...." , "print.gif", "Print", "button" }
Above has the following syntax – {Widget ToolTip, Icon, Widget text,Type of Widget (e.g. button, entry etc)}

As for menubars, the cascading menuitems within a menu-button would be supported as follows:
*$toolBarManObject Register_Window $windowName $menuButtonName *
"$firstlevelMenubuton,$secondlevelMenubutton, $menuItem" "CALLBACK FUNCTION"
there is no limit to upper the number of levels of cascading

For a normal menu-item that is not cascaded should be captured while specifying the widget properties.

set widgetProps($menuButtonName,$widgetName) {"Test-Setup….", "0", "normal", "MenuItemValidfor Data Test_Setup"}
Above has the following syntax – { Widget Text, Underlying Alphabet Index, State[**normal, cascade**], Function to be called for widget validity [optional]}

In order to support dynamic creation widgets the *widgetProps* has special item that checks whether the widget needs to be created under the present set of conditions.

2. APIs provided to return/set the current value in the widget [*GetWidgetVal/SetWidgetVal*] :: The onus is on managers to set the value/return the current value residing inside a widget. That

will save the various windows the burden of managing the variables themselves. Here you just need to pass the "dock-bar-name" [or menubutton-name for menubar manager], "window-name" and "widget-name" to get the value.

set Val [$toolBarManObject getWidgetVal $windowName $dockBarName $buttonName]

set Val [$toolBarManObject setWidgetVal $windowName $dockBarName $buttonName Value]

It is important to mention here that the managers first find the state of the window and set or get the value of the widget associated with that window appropriately. So the "text" widget, for example, that is registered to the particular docked window(s) [or *global* text widget] can take values independently of the "text" widget that is registered to another window but in undocked state [or *local* text widget]. However, when this undocked window is docked, the local "text" widget is destroyed and the undocked window starts using the *global* text widget. We can see a good example here as to how sharing helps in managing the widgets better and in better utilization of precious GUI real estate.

3. *Initializing the value*: The widget can be initialized to some set values based on the window/context while invoking an application. An example could be a *checkbutton* that is "registered" to 2 different windows. One component wants the state of this "*checkbutton*" as "1" whereas the other as "0". So, we read in this initial value in "ToolBarManager" and set the state of the widget accordingly. It shall be handled in ToolBarManager as

set internVar ::$windowName::$widgetVal

4. *Dynamically updating the "managers" on the fly*: The "managers" provide methods that can be called on the fly to update the state of various widgets that are present inside the *toolbar, menubar or statusbar,* including creation of new widgets dynamically. Such an operation would enable/disable certain widgets and/or create new widgets such as menu items for certain *menubuttons* are created if it is valid under the present situations. These optimal-ties help in the maximum utilization of the real estate, avoiding clutter, and share-ability among the various component of the GUI.

It is important to mention here that the managers have been written modularly in that they could be adopted easily by any GUI application, with minor modifications. Further, such "managers" can be added into the existing set of TK widgets, in case of need.

Bibliography

TCL/TK wiki, http://wiki.tcl.tk