Chapter

26 Changing Widgets at Runtime

This chapter contains information on changing VisualWorks widgets while the application is running. For example, it describes how to change labels, how to disable action buttons, and how to make fields invisible. It is by no means a complete description of even this small aspect of VisualWorks, but it should give you some ideas of things you can do in your application and places to look when you want to do more.

When we build a user interface, we place *widgets* on the canvas: action buttons, input fields, labels, etc. Each widget has a *controller* that handles input to the widget. The widget also has a wrapper that handles how the widget is presented to the user. The User Interface *builder* keeps track of all these wrappers, referring to them as *components*. Let's take a quick look at these four aspects of building an application.

The builder

Every subclass of *ApplicationModel* (ie, every window-based VisualWorks application) has a user-interface builder. This builder knows how to construct the user-interface, but, more importantly for us here, it keeps track of all the components of the user-interface. Within the subclass of *ApplicationModel*, the builder can always be referenced as:

```
builder := self builder.
```

Components

The canvas that you construct consists of a group of components. The component is actually a *wrapper* around the widget that you added (such as an action button, a label, an input field, etc.) To reference one of these components, it has to have an ID, which you specify in the ID field in the Properties Tool. I tend to give the component an ID that describes the type of the widget. For example, an action button might have the ID *saveAB*, where the AB specifies that it is an action button. Similarly, an input field might have an ID of *employeeNameIF*. To get the component, our application model asks its builder for the component at the ID we are interested in. For example,

```
component := self builder componentAt: #employeeNameIF.
```

Since the component is a wrapper around a widget, you will sometimes see the above code written as:

```
wrapper := self builder componentAt: #employeeNameIF.
```

Components/wrappers understand how to make themselves visible and invisible, or enabled and disabled. You can tell them to take the keyboard focus, you can change their label strings, and you can set their colors (via their look preferences). We'll see examples of these later.

Widgets

The widget is the object that you think of when you talk about things such as action buttons, input fields, and labels. For active widgets such as these, the widget is actually a view, such as an ActionButtonView. The component is the wrapper around the widget. To get the widget, you simply send the `widget` message to the component. For example,

```
widget := (self builder componentAt: #employeeNameIF) widget.
```

If you inspect a widget that is a view, you'll notice that it has instance variables for its model and controller.

Controllers

Each widget has a controller that handles the keyboard and mouse input. In particular, for input fields, the controller handles the text in the field, the cursor position, and the selection. To get the controller, you send the `controller` message to the widget. For example,

```
controller := (self builder componentAt: #employeeNameIF) widget
controller.
```

If you use the Properties tool to set up a notification or validation method and specify a keyword method with one colon, the parameter to the method will be the controller. So, rather than having to get the controller via something like the above, you will be passed it directly as a parameter. You can then get the new text directly from the controller. For example, your validation method might look like:

```
MyClass>>validateName: aController
   newText := aController text.
   .... do some validation then return true or false...
```

Another thing to be aware of in validation methods is how to discover if user input has been accepted. If you set up an input field to be, say, a date or time, VisualWorks will flash the field if the data is invalid, but will let the user proceed. To tell if the data is valid (ie, it was successfully converted), send the controller the message `hasEditValue`, which returns a Boolean (you can also send this message to the widget). For example,

```
^aController hasEditValue
   ifTrue: [true]
   ifFalse:
      [Dialog warn: 'Invalid date'.
      false]
```

Before we leave the topic of validation, one thing I find frustrating about the validation mechanism is that there is no way to know which widget the user wants to pass focus to when you are in a validation routine. For example, suppose you have a field that requires valid input, such as a date field. If the user can't figure out what to enter and they press the Cancel key, the application still tries to validate the input. So the user can't cancel until they either enter valid data or blank out the field. One solution is to modify *KeyboardProcessor*. Add an instance variable called, say, *controllerRequestingFocus*, and a get accessor for it. In the method `KeyboardProcessor>>requestFocusFor:`, add the following line just before sending the `requestFocusOut` message to the current consumer.

```
controllerRequestingFocus := aController.
```

Now, in your validation routine you can have something like the following. Each validation method may want to check if the cancel key has been pressed, so we break out that code into a separate method. There is an example of this in the file `focus.st`.

```
MyApplication>>validateDate: aController
   self cancelPressed ifTrue: [^true].
   ^aController hasEditValue
      ifTrue: [true]
      ifFalse:
         [Dialog warn: 'Please enter a valid date'.
         false]

MyApplication>>cancelPressed
   ^self builder keyboardProcessor controllerRequestingFocus ==
      (self builder componentAt: #cancelAB) widget controller
```

Modifying things

Now that we have a small amount of background, we can go ahead and show the code for doing various runtime changes. Examples of this can be found in the file `widgets.st` on the diskette. This first example is the class *WidgetAttributes*.

Enabling and Disabling

To enable and disable a widget we send the `enable` and `disable` messages to the component. When a widget is disabled, it appears grayed out to indicate that it can't be used. For example,

```
(self builder componentAt: #saveAB) enable.
(self builder componentAt: #saveAB) disable.
```

Visible and Invisible

We make a widget visible or invisible by sending the messages `beVisible` and `beInvisible` to the component. For example,

```
(self builder componentAt: #salaryIF) beVisible.
(self builder componentAt: #salaryIF) beInvisible.
```

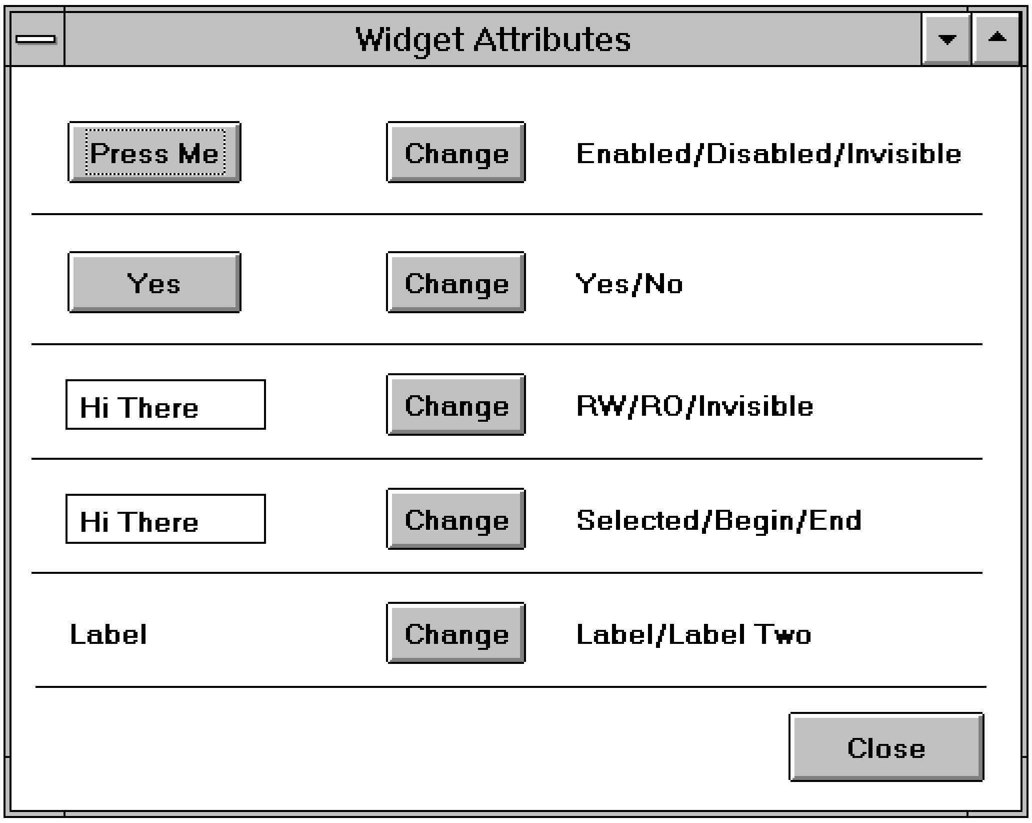

Figure 26-1. Dynamically modifying widget attributes.

Changing labels

To change the label string on a widget, we send the `labelString:` message to the component. For example,

```
(self builder componentAt: #actionAB) labelString:
   (mode == #edit)
      ifTrue: ['Save']
      ifFalse: ['Add'].
```

Unfortunately, getting the label string is not quite as easy. We have to get it from the widget itself via several message sends. For example,

```
labelString := (self builder componentAt: #actionAB) widget label
text asString.
```

Changing selections and cursor position

Selections and cursor position are associated with controllers, so to manipulate them we need to send messages directly to the controller. To make a widget be the active widget, send the `takeKeyboardFocus` message to the controller. In the case of an input field, this will also select and highlight all the text in the field. For example,

```
controller := (self builder componentAt: #salaryIF) widget
controller.
controller takeKeyboardFocus.
```

If you send `takeKeyboardFocus`, you don't need to specifically select the text. However, you can alternatively select a range of characters, or position the cursor at a particular location. Here are some examples,

```
controller selectFrom: 3 to: 5.
   "Select and highlight from position 3 to 5"
controller selectAt: 1.
   "Position the cursor at the beginning"
controller selectAt: controller text size + 1.
   "Position the cursor at the end"
controller find: 'and' startingAt: 1.
   "Position the cursor before 'and' "
controller findAndSelect: 'and'.
   "Find the word 'and' after the current selection, and
   highlight it"
```

Two additional messages, `select` and `deselect`, highlight and unlighlight the selected text. They do *not* change the selection. To retrieve the currently selected text, send `selection` to the controller.

Changing colors and fonts

In the following example class, *WidgetText*, we look at changing colors, fonts, and emphases. This class can be found in the file `widgets.st`.

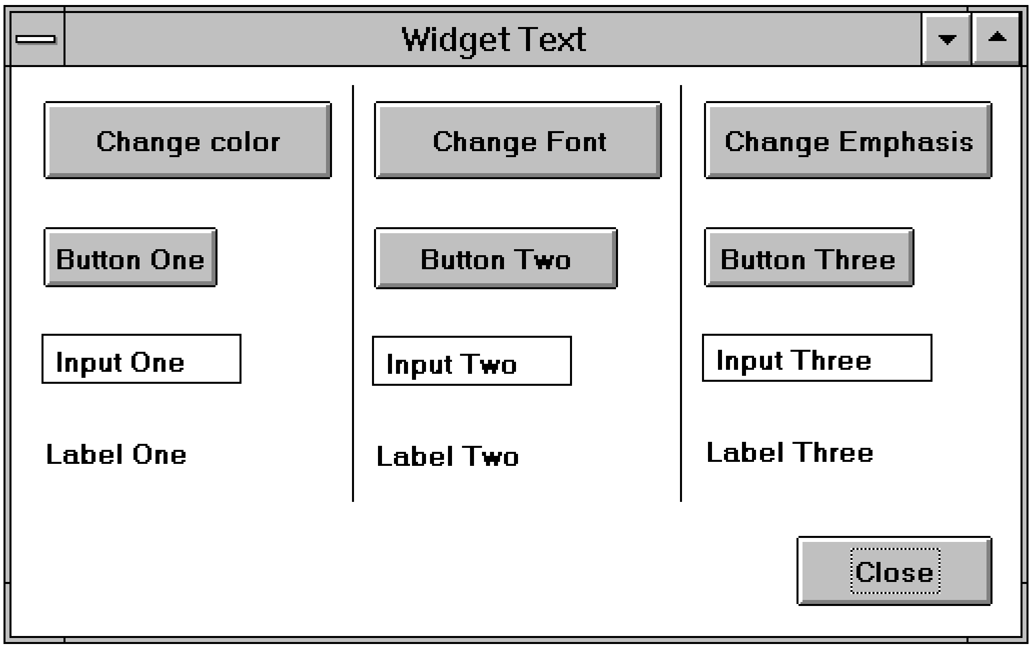

Figure 26-2. Dynamically modifying widget text.

There are several aspects to changing colors and fonts. To change *colors* such as the background colors and foreground colors of widgets, we have to get the component's look preferences (actually, the `lookPreferences` messages returns a copy of the look preferences). We then set the colors appropriately and give the new look preferences to the component. The `lookPreferences:` message tells the component that the look prefences have changed, so the component redisplays itself. For example,

```
component := self builder componentAt: #saveAB.
lookPrefs := component lookPreferences.
lookPrefs setForegroundColor: ColorValue green.
component lookPreferences: lookPrefs.
```

To change the *text attribute* of a widget, we create an instance of *TextAttributes*. We'll use a default style named *#large*, although you get more flexibility by creating an instance of *TextAttributes* using *CharacterAttributes.* We replace the style rather than modifying the one we have because it may be one that this widget shares with other widgets. If we changed the style directly, the other widgets would also get the change. Note that we invalidate the widget afterwards, which causes it to redisplay itself.

```
widget := (self builder componentAt: #saveAB) widget.
widget textStyle: (TextAttributes styledNamed: #large).
widget invalidate.
```

To change the *text emphasis* for widgets with labels, such as an action button or a label, we can do something like the following. We get the label text and emphasize it by passing either an array of emphases or a single emphasis. Then we set the label's text to be the newly emphasized text.

```
widget := (self builder componentAt: #saveAB) widget.
emphasis := Array
   with: #italic
   with: #large
   with: #color -> ColorValue pink.
newText := widget label text emphasizeAllWith: emphasis.
widget label text: newText.
widget invalidate.
```

If we want to emphasize text in an input field, the approach is very similar, but we get the text from the controller rather than from the label. For example,

```
widget := (self builder componentAt: #nameIF) widget.
widget editText: (widget controller text emphasizeAllWith: #(#italic
#large).
widget invalidate.
```

A shortcut for bold text is to send the message `allBold` to the text. Note that in the *WidgetText* class provided in `widgets.st`, the label fields have been lengthened. Without the lengthening, the old text is not completely cleaned up when the text size changes to a smaller size.

Changing Menus

In this section, we'll take a look at three different kinds of menus: menu bars, menu buttons, and text field menus. For each of these menu types we want to do three different things. First, we want to replace the menu with a completely different menu based on some event. Second, we want to dynamically select the menu to present when the user goes to select a menu item. Third, we want to modify the menus, disabling menu items, hiding items, changing colors, and adding items. Examples of the following can be found in *MenuDynamicModify* and *MenuDynamicCreate* in the file `widgets.st`. The first example, *MenuDynamicModify*, shows various modifications you can make to a menu.

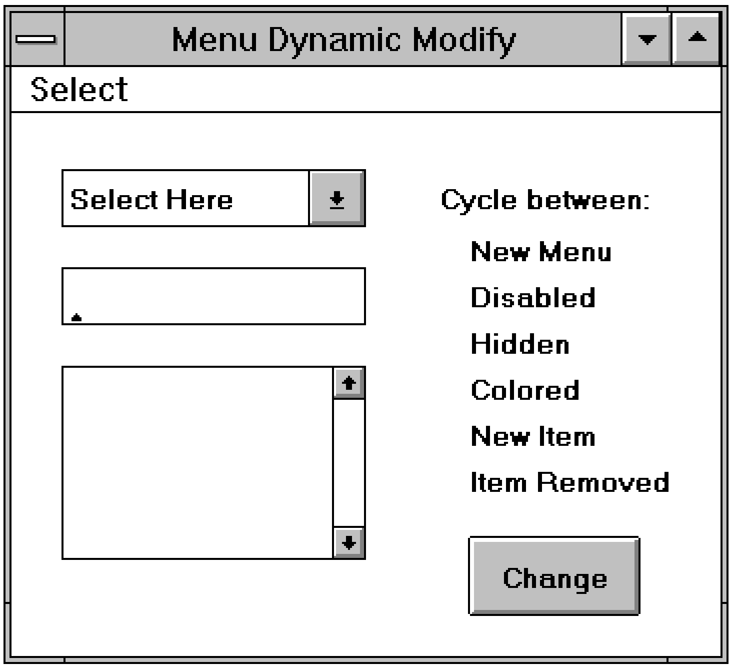

Figure 26-3. Dynamically modifying menus.

Replacing menus

When we create a menu, we specify a method in the Menu field in the Properties Tool. Usually this method is invoked once by the builder, returns a menu, and is never invoked again. To make it possible to replace the menu, we instead specify a method that returns a *ValueHolder*. The *ValueHolder* holds a menu, and by sending the `value: aNewMenu` message to the *ValueHolder*, we can replace the menu completely. For example, in the `initialize` method we might have something like:

```
initialize
   menuButton := nil asValue.
   inputField := String new asValue.
   menuButtonMenuVH := self menuOne asValue.
   inputFieldMenuVH := self menuOne asValue.
   MenuBarVH := self class menuOne asValue.
```

Then at any point in the program, we can install a new menu by doing the following.

```
   menuButtonMenuVH value: self menuTwo.
   inputFieldMenuVH value: self menuTwo.
   MenuBarVH value: self class menuTwo.
```

Modifying menus and menu items

To modify a menu or menu item, we first need to get the menu (for the menu bar, I am assuming that we are modifying one of the pull-down menus). There are two basic ways to get the menu from the builder. One way gets it via the component: from the widget for a *menu button*, from the controller for a *text field*, and for a menu bar we get the submenu by specifying the main menu item, then asking for its submenu.

```
menu := (self builder componentAt: #departmentMB) widget menu.
menu := (self builder componentAt: #nameIF) widget controller menu.
menu := (MenuBarVH value menuItemLabeled: 'Select') submenu.
```

The other way of getting the menu is via the method specified in the *Menu* field of the Properties Tool. If the method gives back a menu, then we do the first line below. If the method gives back a *ValueHolder*, we do the second line. In both cases the parameter to `menuAt:` is the name we specified in the *Menu* field of the Properties Tool.

```
menu := self builder menuAt: #menuButtonMenu.
menu := (self builder menuAt: #menuButtonMenuVH) value.
```

Once we have the menu, we get the menu item we want to change by sending the `menuItemLabeled:` message (we can also get the item by specifying its index or the value or selector associated with the item). For example,

```
menuItem := menu menuItemLabeled: 'One'.
menuItem := menu menuItemAt: 3.
menuItem := menu menuItemWithValue: aValueOrMethodSelector.
```

Here are some of the changes you can make to a menu or a menu item while your application is running.

`menuItem disable.`	Disable a menu item
`menuItem enable.`	Enable a menu item
`menu hideItem: menuItem.`	Hide a menu item
`menu unhideItem: menuItem.`	Make visible a hidden menu item
`menu backgroundColor: ColorValue green.`	Change the menu color
`(menu menuItemLabeled: 'Three') color: ColorValue red.`	Change the color of a menu item
`menu addItemLabel: 'New Item' value: #someMessageSelector.`	Add a menu item to the end
`menu removeItem: (menu menuItemLabeled: 'New Item').`	Remove a menu item

Creating the menu at selection time

We can dynamically choose the menu to display when the user goes to select a menu item. Before any items are shown, we figure out what the menu should look like and display the newly created menu. To make this work, we install a block of code in the `preBuildWith:` method, rather than creating a method that will return the menu. This block of code will be executed each time the user tries to select the menu. (Note that this technique does not work for menu bars.) An example of this technique can be seen in the class *MenuDynamicCreate* in the file `widgets.st`.

```
preBuildWith: aBuilder
   aBuilder menuAt: #menuButtonMenu put: [self selectMenu].
   aBuilder menuAt: #inputFieldMenu put: [self selectMenu].
   aBuilder menuAt: #textEditorMenu put: [self selectMenu]

selectMenu
   mode == #menuOne
      ifTrue: [^self menuOne]
      ifFalse: [^self menuTwo]
```

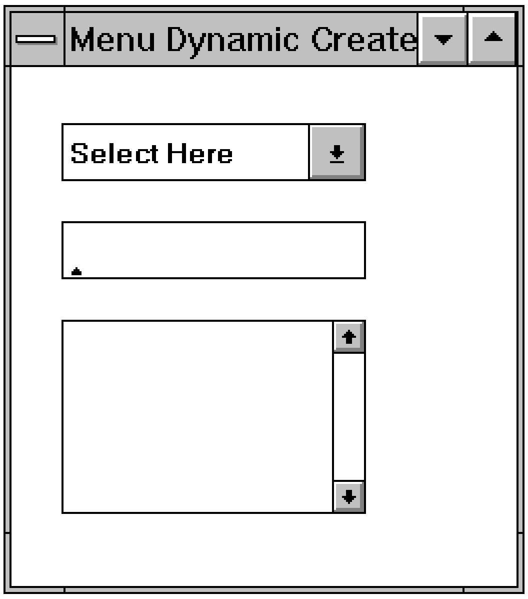

Figure 26-4.
Dynamically creating menus.

Keyboard events and double-clicking

All active widgets (widgets that accept keyboard input) allow you to intercept keystrokes by specifying a block of code as a keyboard hook. List and Table widgets also allow you to specify actions to take when the user double clicks a selection using a mouse. You specify the double click method in the Notification page in the Properties Tool. Alternatively, you can programmatically specify a block of code or a method to invoke, which overrides any method you specified in the Properties Tool.

Let's look at an example using a List box (you can find the code in the file `listdemo.st` on the diskette). Create an application class called *ListDemo* with two instance variables, *list* and *keyboardSelectors*.

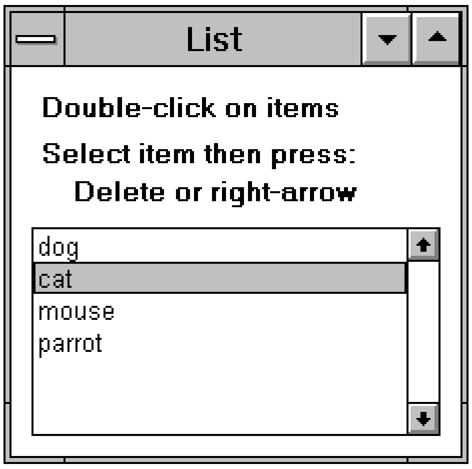

Figure 26-5.
The ListDemo window.

We'll specify the double-click action in two ways, with a method and a block of code to execute. We'll write code to delete the selected item when the user presses the Delete key, and to show a dialog box when the user presses the right arrow key. To make the keyboard hook general, we'll record the the keys and the methods they should invoke in a Dictionary. We set this up during initialization.

```
ListDemo>>initialize
   pets := SelectionInList new.
   pets list: #('dog' 'cat' 'mouse' 'parrot') asList.
   self mySetUpKeyboardSelectors

ListDemo>>mySetUpKeyboardSelectors
   keyboardSelectors := Dictionary new.
   keyboardSelectors
      at: Character del put: #deleteSelection;
      at: #Right put: #expandSelection
```

Here are the methods that will be executed when the user presses the delete key or the right arrow key.

```
ListDemo>>deleteSelection
   | collection index |
   index := self pets selectionIndex.
   index > 0 ifTrue: [self pets list removeAtIndex: index]

ListDemo>>expandSelection
   self pets selectionIndex > 0
      ifTrue: [Dialog warn: 'Details of selection']
```

Now we want to specify a keyboard hook that will allows us to intercept keystrokes. We'll also specify a block of code that will be executed when the user double clicks on a list selection. We tell the controller about the keyboard hook and the controller's dispatcher about the double-click block or method.

```
ListDemo>>postBuildWith: aBuilder
   | listController |
   super postBuildWith: aBuilder.
   listController := (self builder componentAt: #petsLB) widget
controller.
   listController keyboardHook: self myKeyboardHookBlock.
   listController dispatcher doubleClick: [self blockDoubleClick].
```

A keyboard hook is a block of code taking two parameters, the event and the controller. It should return the keyboard event or nil, nil meaning that no further processing should be done on the keyboard event. We'll always return the event so that the list can do its usual processing. Because we store the actions in a Dictionary, we can change the keystrokes that we are looking at, and the actions they should perform, based on the current context.

```
ListDemo >myKeyboardHookBlock
   ^
   [:event :controller |
   | selector |
   selector := keyboardSelectors at: event keyValue ifAbsent: [nil].
   selector notNil ifTrue: [self perform: selector].
   event]
```

When the block is executed in response to a double mouse click on a selection, we'll change the double click mechanism to specify a method to be executed rather than a block. Then when the method we specify is invoked, we'll change the mechanism back to using a block.

```
ListDemo>>blockDoubleClick
   Dialog warn: 'Block double click'.
   (self builder componentAt: #petsLB)
      widget controller dispatcher doubleClickSelector:
#selectorDoubleClick

ListDemo>>selectorDoubleClick
   Dialog warn: 'Selector double click'.
   (self builder componentAt: #petsLB)
```

```
      widget controller dispatcher doubleClick: [self
blockDoubleClick].
```