Chapter

31

Customizing your Environment

One of the things that makes Smalltalk development so much fun is that you can easily modify the development environment. If you don't like something, change it. This chapter contains examples of such changes. They are preferences of mine but not necessarily those of everyone (if they were so good, they'd be part of the standard image). To make the changes part of your standard environment, you can either save the image with your changes, or file the changes in every time you start up the image. For more information on filing in changes, see Chapter 33, Managing Source Code.

Keyboard

In VisualWorks the delete key and the backspace key work identically, both deleting the character to the left of the cursor. To change the delete key so it deletes to the right of the cursor, evaluate the following.

```
LookPreferences deleteForward: true.
```

Another change I like to make is to map ctrl-X, ctrl-C, and ctrl-V to Cut, Copy, and Paste, as is standard in Windows applications. I also map ctrl-A to Accept. One way to remap your keyboard is to modify the class side method `initializeDispatchTable` in `ParagraphEditor`. If you make the following changes, you must execute `ParagraphEditor initializeDispatchTable` after accepting the method. This is most easily done by selecting the text in quotes at the top of the method and executing *do it*. The will take effect in the next Browser you open, so close your current Browsers and open new ones.

```
ParagraphEditor class>>initializeDispatchTable
   ....
   Keyboard bindValue: #cutKey: to: Ctrlx.
   Keyboard bindValue: #copyKey: to: Ctrlc.
   Keyboard bindValue: #pasteKey: to: Ctrlv.
   Keyboard bindValue: #acceptKey: to: Ctrla.
   ....
```

This approach is a reasonable one if everyone will be getting the same keyboard mapping. An alternative way, which allows developers to easily customize their own keyboard maps, is to create a new class method,

`keyboard`, for ParagraphEditor that returns the class variable, *Keyboard*. Once you have this method, you can modify the keyboard bindings by sending `bindValue:to:` messages to the keyboard. This scheme has the advantage that programmers don't have to change the `initializeDispatchTable` method, or reinitialize it. As before, the changes will only take effect when new Browsers are opened. Here's some code showing how this approach might work.

```
ParagraphEditor class>>keyboard
   ^Keyboard

ParagraphEditor keyboard
   bindValue: #cutKey: to: (TextConstants at: #Ctrlx);
   bindValue: #copyKey: to: (TextConstants at: #Ctrlc);
   bindValue: #pasteKey: to: (TextConstants at: #Ctrlv);
   bindValue: #acceptKey: to: (TextConstants at: #Ctrla).
```

The wonderful thing about Smalltalk is that there are many ways to do things. If you don't want to add methods to system classes, there is yet another approach to take. We can directly get hold of the *Keyboard* class variable, then send it the relevant messages. For example,

```
(ParagraphEditor classPool at: #Keyboard)
   bindValue: #cutKey: to: (TextConstants at: #Ctrlx).
```

Ctrlx, etc., are found in the TextConstants pool dictionary. ParagraphEditor has specified TextConstants as a pool dictionary, but if you evaluate the above line in a workspace you have to specify where to find Ctrlx. At this point we are not quite done, because by default ctrl-C is the interrupt key, which takes precedence over Copy. We need to remap the interrupt key to something else. I choose ctrl-Q because it is easily remembered (Q = quit).

```
InputState interruptKeyValue: (TextConstants at: #Ctrlq).
```

If you don't like to remap ctrl-C but would still like these changes, and alternative technique would be to map the keys to a double key combination. For example, we might map Cut, Copy, and Paste to Esc-x, Esc-c, Esc-v. Here's an example of doing this, assuming we have a `keyboard` method.

```
ParagraphEditor keyboard
   bindValue: #cutKey: to: (TextConstants at: #ESC) followedBy: $x;
   bindValue: #copyKey: to: (TextConstants at: #ESC) followedBy: $c;
   bindValue: #pasteKey: to: (TextConstants at: #ESC) followedBy: $v.
```

Besides these changes, I also remap the Replace and Search keys and include several emacs key bindings such as ctrl-D for delete-character, ctrl-K for delete-to-end-of-line, and ctrl-E for end-of-line. The emacs key-bindings code is available from the Smalltalk Archives (see Chapter 35, Public Domain Code and Information). If you have a Sun keyboard, you can map the keys on the left by using #L1, #L2, etc., as the names. For example,

```
Keyboard bindValue: #copyKey: to: #L6.
Keyboard bindValue: #pasteKey: to: #L8.
Keyboard bindValue: #cutKey: to: #L10.
```

VisualWorks 2.5 change

In VisualWorks 2.5, the TextEditorController gets its own copy of the keyboard table from ParagraphEditor, its superclass. If you make changes to ParagraphEditor's keyboard table, you will need to reinitialize TextEditorController after making the changes and before opening new windows.

```
TextEditorController initialize
```

Launcher

There are several changes I like to make to the Launcher. However, rather than changing the class *VisualLauncher*, we'll create a *MyVisualLauncher* as a subclass of VisualLauncher. Here are some examples of what we might add to it. Because the launchers are created differently in VisualWorks 2.0 and 2.5, we'll show the differences where appropriate. (You can find the source code in the files `launch20.st` and `launch25.st`.) After making the changes, open a MyVisualLauncher and close the old VisualLauncher. Figure 31-1 shows the customized Launcher.

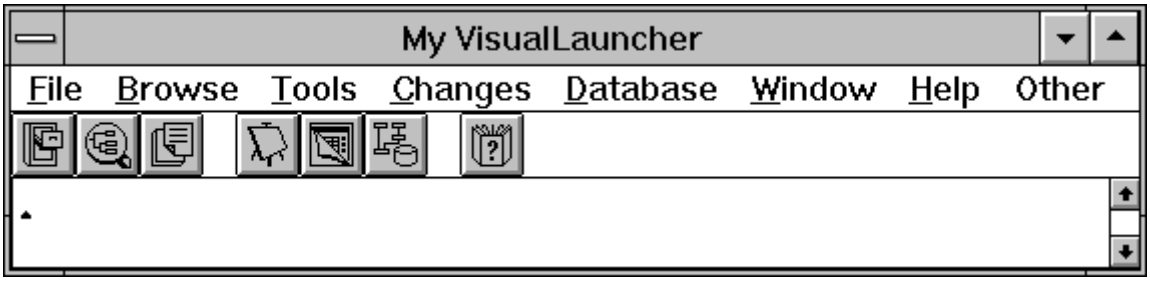

Figure 31-1. The customized Launcher.

Window label

The first thing we'll do is modify the window label so that it's obvious we're not using a standard Visual Launcher. We change the label after the window is built, but before it is displayed.

```
MyVisualLauncher>>postBuildWith: aBuilder
   super postBuildWith: aBuilder.
   aBuilder window label: 'My VisualLauncher'
```

Other menu

Next we add a new menu to the menu bar, the *Other* menu. This is where we will put most of our additional actions that we'd like to invoke from the Launcher. The Other menu will look like the following.

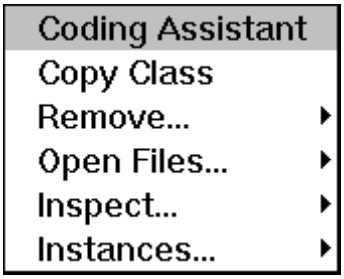

Figure 31-2.
The Other menu.

We'll add the new Other menu to the end of the menu bar, which we do by overriding the `newMenuBar` method and adding a new menu item to the end.

```
MyVisualLauncher>>newMenuBar
   | menuItem |
   menuItem := MenuItem labeled: 'Other'.
   menuItem submenu: self class otherMenu.
   ^(super newMenuBar) addItem: menuItem; yourself
```

VisualWorks 2.0

Copy the `browseMenu` method in the class side `resources` protocol of VisualLauncher to the new MyVisualLauncher class. Modify the name to `otherMenu` both in the method name *and* in the comment. Accept the method, then highlight the text inside the comment and *do it*, which brings up a menu editor.

Modify the menu text and method names to look like the following then press the Build button followed by the Install button. Press OK when given the Install screen. Note that a Tab separates the menu item name from the method name. What follows are some items that I like to include in my Other menu. Obviously you can add your own favorite actions and modify or remove the ones shown here.

```
Coding Assistant  otherOpenCodingAssistant
Copy Class     otherCopyClass
Remove...         nil
   Window      otherRemoveWindow
   Dialogs     otherRemoveDialog
Open Files...     nil
   Inspect     otherInspectOpenFiles
   Close    otherCloseFile
Inspect...        nil
   Smalltalk      otherInspectSmalltalk
   Undeclared  otherInspectUndeclared
Instances...      nil
   Inspect     otherInspectInstances
   Remove      otherRemoveInstances
```

VisualWorks 2.5

In VisualWorks 2.5 you can directly edit the menu bar using the menu editor, placing the *Other* menu wherever you think appropriate. However, you must first select Settings from the File menu, then choose *Use Enhanced Tools* in the UI Options tab. If you don't use the Enhanced Tools, the menu will not be correctly built and you will get an exception when you open the new Launcher.

However, we will add the Other menu to the end of the menu bar without modifying the `menuBar` method, which allows us to inherit the VisualLauncher menu bar. In the ResourceFinder, highlight MyVisualLauncher then select New Menu from the Resources menu. Add the items shown above then install your changes by selecting Install from the Menu menu.

Both

Moving to the instance side of MyVisualLauncher, in a new `actions` protocol write the following methods. Once you've added this code, open a new Launcher by executing `MyVisualLauncher open` in a workspace, and close the old Launcher.

```
MyVisualLauncher>>otherInspectSmalltalk
```

```
   Smalltalk inspect

MyVisualLauncher>>otherInspectUndeclared
   Undeclared inspect

MyVisualLauncher>>otherOpenCodingAssistant
   CodingAssistant open
```

The following methods inspect and close open files. It's possible to lose a reference to a file without having closed it, especially if an exception is raised in the file reading or writing code. The garbage collector will collect your file object, but won't close the file on disk.

```
MyVisualLauncher>>otherInspectOpenFiles
   (ExternalStream classPool at: #OpenStreams) copy inspect

MyVisualLauncher>>otherCloseFile
   | path count |
   count := 0.
   path := Dialog request: 'Full path for file you want to close?'
initialAnswer: ''.
   (ExternalStream classPool at: #OpenStreams) copy
      do: [:each | each name = path
         ifTrue:
            [each close.
            count := count + 1]].
   Dialog warn: 'Closed ' , count printString , ' files'
```

The next methods remove windows that are on the screen but which you can't remove the normal way. Sometimes windows won't close, and sometimes dialogs are left orphaned if the code associated with them raises an exception.

```
MyVisualLauncher>>otherRemoveDialog
   "Close orphan dialogs"
   ScheduledControllers scheduledControllers
      do: [:each | (each isKindOf: ApplicationDialogController)
         ifTrue: [each closeAndUnschedule]]

MyVisualLauncher>> otherRemoveWindow
   | label |
   label := Dialog request: 'Label for window you want to remove?'
initialAnswer: ''.
   ScheduledControllers scheduledControllers
      do: [:each |
         (each view label = label or: [each view label = label
asSymbol])
            ifTrue: [each closeAndUnschedule]]
```

The next methods inspect and remove instances of a class. Sometimes you will have instances that won't disappear. This indicates a problem with your application in that some object is still referencing the instance. However, you may still want to get rid of the instances. Be careful of this feature though; if you remove instances of system classes such as OrderedCollection, your system will no longer function correctly, if at all. (Also, don't plan on saving the image because there is something wrong with your application and it is not releasing objects correctly.) We use the `become:` message to change each instance of the class to a new instance of String. In VisualWorks, `become:` swaps all references to the receiver and to the argument. So, every object that referenced the instance of the specified class now references a new instance of String. Also,

every reference to this new instance of String now holds onto the instance of the specified class. Since the string is new, there are no objects holding onto it, so your instances will now be garbage collected. (If you try and make something become *nil*, you will swap the references so that every place that referred to *nil* will now refer to the instance of the specified class, which will cause untold problems.)

```
MyVisualLauncher>>otherInspectInstances
   | class className |
   className := Dialog request: 'Inspect instances of which class?'
initialAnswer: ''.
   class := Smalltalk at: className asSymbol ifAbsent: [nil].
   class notNil
      ifTrue: [class allInstances inspect]
      ifFalse: [Dialog warn: 'This class does not exist']

MyVisualLauncher>>otherRemoveInstances
   | class className count |
   className := Dialog request: 'Class for which to remove
instances?' initialAnswer: ''.
   class := Smalltalk at: className asSymbol ifAbsent: [nil].
   class notNil
      ifTrue:
         [count := class allInstances size.
         class allInstancesDo: [:each | each become: String new].
         Dialog warn: 'Removed ' , count printString , ' instances']
      ifFalse: [Dialog warn: 'This class does not exist']
```

Sometimes you will decide that a class has too much responsibility and will split it into two classes. The easiest way to do this is to start with two identical classes and remove the methods that are not needed. One way to create a copy of the class is to file it out, change the class name, then file the original class back in (in Envy, you can't rename classes so you'll have to change the name in the disk file). Another approach is to add a menu item to the Launcher.

```
MyVisualLauncher>>otherCopyClass
   | sourceName sourceClass destinationName writeStream newSourceCode
|
   sourceName := Dialog request: 'Class to Copy?'.
   sourceClass := Smalltalk at: sourceName asSymbol ifAbsent:
      [^Dialog warn: 'This class does not exist'].
   destinationName := Dialog request: 'New Class?' initialAnswer: ''.
   destinationName = '' ifTrue: [^self].
   (Smalltalk at: destinationName asSymbol ifAbsent: [nil]) notNil
      ifTrue: [^Dialog warn: 'This class already exists'].
   writeStream := (String new: 1000) writeStream.
   sourceClass fileOutSourceOn: (SourceCodeStream on: writeStream).
   newSourceCode := writeStream contents
      copyReplaceAll: sourceName
      with: destinationName.
   newSourceCode readStream fileIn
```

Changes

Let's make a couple of changes to the Changes menu. I like to have *Inspect Change Set* at the top, and I like to add a *Condense Changes* option. All the code that you write is compiled into byte-codes and these byte-codes stay in the image when you save it. However, the source code goes into the change file, which records all the modifications to classes and methods. Because the change file grows as you make changes to your application,

it's a good idea to periodically condense it, removing all but the most recent version of your code. (Alternatively, you could periodically start with a fresh image and save it under your name.) Here's what the new Changes menu will look like.

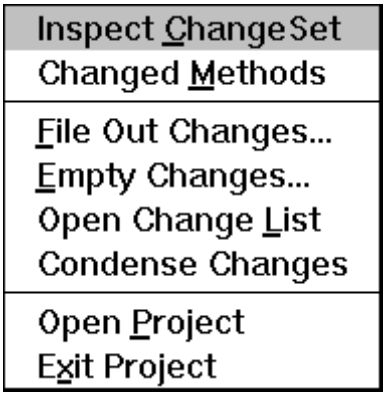

Figure 31-3.
The Change Menu.

VisualWorks 2.0

For VisualWorks 2.0, go to the `resources` protocol on the class side and copy `changesMenu` from VisualLauncher to MyVisualLauncher. In the `changesMenu` method, highlight the text inside the command, and *do it*. Change the order of the menu items to put Inspect Change Set at the top, then add the entry shown below.

VisualWorks 2.5

For VisualWorks 2.5, you'll need to make sure that you have selected Enhanced Tools in the Settings. Copy over the `menuBar` method from the VisualLauncher class side `resources` protocol to MyVisualLauncher. Change the order of the menu items to put Inspect Change Set at the top of the Change menu, then add the entry shown below.

Both

For both VisualWorks 2.0 and 2.5, add the following to the Changes menu.

```
Condense Changes  changesCondenseChanges
```

Now we need to write the `changesCondenseChanges` method in the instance side `actions` protocol and open a new Launcher.

```
MyVisualLauncher>>changesCondenseChanges
   SourceFileManager default condenseChanges
```

The `condenseChanges` message removes all but the most recent versions of the methods and classes in the changes file. A similar message, `condenseChangesOntoSources`, adds the most recent changes to the sources file (visual.sou), removing *all* changes from the changes file. This latter message is useful after you get a new release of the software and you want to add code that will be part of the base image, such as the Advanced Tools. If you add the new code then do `SourceFileManager default condenseChangesOntoSources`, the changes will be added to the source file and will be available to everyone. You could certainly leave the changes in the change file, but then everyone would get a larger change

file than necessary. In regular use you definitely do *not* want developers to condense their changes onto the source file. General usage should be to condense changes within the changes file.

If we simply wanted to add a group of menu items to the end of the menu, we wouldn't need to override the `changesMenu` method. Instead we could add items to the end of the menu by adding more code to the `newMenuBar` method.

```
MyVisualLauncher>>newMenuBar
   | menuItem menuLabels menuSelectors changesMenu newMenuBar |
   newMenuBar := super newMenuBar.
   menuItem := MenuItem labeled: 'Other'.
   menuItem submenu: self class otherMenu.
   newMenuBar addItem: menuItem.
   menuLabels := #('Condense changes').
   menuSelectors := #(#changesCondenseChanges).
   changesMenu := newMenuBar menuItemLabeled: 'Changes'.
   changesMenu submenu addItemGroupLabels: menuLabels values:
menuSelectors.
   ^newMenuBar
```

Default browser

I prefer to use the System Full Browser rather than the System Browser, so I'd rather have the Browser button bring up a Full Browser. To do this, we simply override `browseAllClasses` as follows.

```
MyVisualLauncher>>browseAllClasses
   self openApplicationForClassNamed: #FullBrowser
```

Adding VisualLauncher menus to a mouse button

If you use a virtual desktop such as mvwm or HP Dashboard®, you may find yourself moving Browsers and other windows onto other virtual screens. Then you may find yourself either moving between desktops whenever you want to use the Launcher, or creating additional Launchers for your other desktops. Below is some code that will add the Launcher menus to the mouse window menu. This is the right hand mouse button on a three-button mouse (eg, for Unix workstations), or Ctrl-Righthand mouse button on a two-button mouse (eg, for PCs). We use the window menu rather than the operate menu because the window menu is defined only in one place, whereas the operate menu is context sensitive and is defined in many places.

```
   windowMenu := StandardSystemController classPool at:
#ScheduledBlueButtonMenu.
   launcherItem := windowMenu menuItemLabeled: 'Launcher' ifNone:
[nil].
   launcherItem notNil ifTrue: [windowMenu removeItem: launcherItem].
   menuItems := MyVisualLauncher new newMenuBar menuItems.
   subMenu := Menu new addItemGroup: menuItems.
   windowMenu addItem:
      ((MenuItem labeled: 'Launcher') submenu: subMenu)
```

You will also need to make a change to *StandardSystemController*, to redirect execution from the controller to a MyVisualLauncher. It's definitely a hack, but it works quite nicely. In `StandardSystemController>>blueButtonActivity`, find the code at the bottom that says `self perform:choice`. Modify this to be the following:

```
(self respondsTo: choice)
   ifTrue: [self perform: choice]
   ifFalse: [MyVisualLauncher new perform: choice]
```

If you are using ENVY, you could extend the window menu by executing:

```
windowMenu := StandardSystemController classPool at:
#ScheduledBlueButtonMenu.
windowMenu menuItemLabeled: 'ENVY'
   ifNone:
      [windowMenu addItem:
         ((MenuItem labeled: 'ENVY') submenu: VisualLauncher
browseMenu)].
```

You would then modify `blueButtonActivity` to read as follows. This works because the ENVY menu returns a block rather than a method to perform.

```
(choice isSymbol)
   ifTrue: [self perform: choice]
   ifFalse: [choice value].
```

Templates

To change the method template (the thing you see if you have a protocol selected but no method selected), modify `sourceCodeTemplate` for class *Behavior*. To put in a date and copyright, do something like the following. If you want anything more sophisticated, it might be clearer to use a Stream to build up the string.

```
Behavior>>sourceCodeTemplate
   ^'methodName
   "Copyright My Company: ', Date today printString, ' ', Time now
printString, '"
   '
```

Another change that you might want to make for methods is to record each modification to a method. The starting point for this is `TextController>>doAccept`, but the modifications are non-trivial.

If you want to change the class comment template so that it gives a specific layout for the comments you want developers to write, you can modify `ClassDescription>>commentTemplateString`. Take a look at the comments for a few of the system classes (such as Date and KeyedCollection) and use them as a guideline for building your own template.

Screen Color

If you want to change the colors of the development environment, the easiest way to do so is to go to the appropriate WidgetPolicy class. If you are using Motif widgets on a Unix workstation, you can modify the class side method `initializeDefaultGenericColors` to specify the colors you want, such as shown below, then execute `MotifWidgetPolicy initializeDefaultGenericColors`.

```
MotifWidgetPolicy class>>initializeDefaultGenericColors
   ...
   matchAt: SymbolicPaint background put: (background := ColorValue
lightCyan);
   matchAt: SymbolicPaint selectionBackground put: ColorValue pink;
```

```
    matchAt: SymbolicPaint selectionForeground put: ColorValue black.
    ...
```

Alternatively, you can change the colors by sending messages rather than modifying a system method, which is more appropriate if you want to change the colors using a fileIn. To do this, send `matchAt:put:` messages to the appropriate object. Again, if it is a Motif widget, you would do the following:

```
MotifWidgetPolicy defaultColorWidgetColors
   matchAt: SymbolicPaint background put: ColorValue lightCyan;
   matchAt: SymbolicPaint selectionBackground put: ColorValue pink.
```

Once you've done this, you need to update the default screen and refresh the display. You can execute the following lines to accomplish this.

```
Screen default updatePaintPreferences.
ScheduledControllers restore.
```

If you are not on a platform that runs X, VisualWorks pays attention to the color scheme you chose for the platform (on Windows, these are the colors specified in the Control Panel). To disable this and make VisualWorks pay attention to the colors you specify using the mechanism above, comment out the line that says `self installPlatformPreferencesFrom: aGraphicsDevice on: colors` in the method `WidgetPolicy class>>defaultWidgetColorsOn:`.

Speed up startup

On a Unix system, VisualWorks takes a long time to come up. One of the reasons is that during startup, VisualWorks looks at every font on the system. When we modified this code to only look at the fonts that are necessary, we found that the startup time went from about thirty seconds down to seven seconds. To do this change, modify the `FontPolicy>>initializeFor:` method. In it, change the following line to the three lines below.

```
availableFonts := aDevice listFontNames collect: [:name | fontClass
parse: name].

availableFonts := false
      ifTrue: [aDevice listFontNames collect: [:name | fontClass
parse: name]]
      ifFalse: [self necessaryFonts collect: [:name | fontClass
parse: name]]
```

To get the list of fonts that should be returned in the `necessaryFonts` method, inspect `Screen default`. From the inspector, inspect *openFonts*. Highlight the array of Associations and move it into the `necessaryFonts` method. You'll need to remove the value part of each association and make the strings into a literal array. You should end up with something like the method shown here.

```
necessaryFonts
   ^#('-adobe-helvetica-medium-r-normal--12-120-75-75-p-67-iso8859-1'
   ...
   '-adobe-helvetica-medium-r-normal--14-140-75-75-p-77-iso8859-1')
```

These changes will take effect the next time you load the image so save the image, exit, and restart VisualWorks. If you want to revert back to reading all the fonts (perhaps you are going to run the image on another type of computer), simply change the *false* to *true* in `FontPolicy>>initializeFor:`.