Chapter

25

Hooks into the System

Smalltalk provides various hooks into the system that allow you to do additional processing on top of the system provided behavior. The two areas that you are most likely to need these hooks are when copying objects and when opening and closing an application with a user interface. However, as you dig through the class library, you will inevitably find additional hooks, and these will inevitably give you ideas on new features you can add to your applications, or new ways of structuring your applications. Since most of this chapter describes user interface hooks, it is located in the User Interface section, despite the more general section on copying.

Copying

postCopy

When you send the `copy` message to an object, it returns a copy of itself. However, it returns what is known as a *shallow copy*. You have a new instance of the object's class, but the instance variables in the copy contain *exactly* the same object as in the original. So, if you change part of one of the instance variables in the copy, the original gets the same change. Let's look at an example. Create two classes, *Person* and *Address,* as shown below (you can find the code in the file `copydemo.st`). Add accessors to all the instance variables. If you then run the code that follows, you will see that the street address of personOne is '221 Jones Court', which is not what you want.

```
Object subclass: #Person
   instanceVariableNames: 'name address '

Object subclass: #Address
   instanceVariableNames: 'streetAddress city '

(address := Address new)
   streetAddress: '916 Smith Avenue';
   city: 'Boulder'.
(personOne := Person new)
   name: 'Alec Sharp';
   address: address.
```

```
personTwo := personOne copy.
personTwo address streetAddress: '221 Jones Court'.
personOne address streetAddress inspect
```

The reason for the unexpected result is that *Object* basically implements `copy` as `shallowCopy`. The comment for `shallowCopy` says "Answer a copy of the receiver which shares the receiver's instance variables." Fortunately, `copy` also provides a hook so that you can do further processing if a shallow copy is not adequate. The actual implementation of `copy` looks like the following.

```
Object>>copy
   ^self shallowCopy postCopy
```

This gives us an opportunity to get the behavior we want by implementing the method `postCopy`, in which we can make copies of the instance variables. When you implement `postCopy`, you should always do `super postCopy` in case one of your superclasses needs to do something. We want to make a copy of the *address* object so that we don't see the behavior shown above. If you implement the `postCopy` shown then rerun the above code, you'll see that personOne still has a street address of '916 Smith Avenue'.

```
Person>>postCopy
   super postCopy.
   address := address copy
```

Other uses for `postCopy` include situations where you need to reset something after copying an object. For example, when you copy an instance of *Model*, the copy does not get the list of dependents.

```
Model>>postCopy
   super postCopy.
   self breakDependents
```

(VisualWorks used to provide a `deepCopy` which recursively copied the instance variables as well. However, the implementation could lead to problems when objects pointed to each other. In *The Journal of Object-Oriented Programming*, September 1994, Wilf Lalonde and John Pugh describe a partial implementation of a `deepCopy`.)

copyEmpty:

While it's not very common, you can create a class that is a subclass of a collection class, and add new instance variables. As you add items to your collection, it may need to grow in size. The way this happens is that a new collection object is created, the collection items are copied across, then the old object *becomes* the new object. Unfortunately, your instance variables are not copied across into the new object. Here's an example that illustrates this. The following code creates an instance of *MyClass* with a size of two, sets the two instance variables, then adds three items to the collection. Since the collection has a size of two, it has to grow. You'll need to write accessors for the variables.

```
Set variableSubclass: #MyClass
   instanceVariableNames: 'varOne varTwo '

coll := MyClass new: 2.
coll varOne: 1.
coll varTwo: 2.
```

```
coll add: 22; add: 33; add: 44.
coll inspect
```

When you inspect the collection, you will find that your instance variables no longer have the original values. Fortunately there is a hook that lets you copy them across. To get them, you must override Collection's `copyEmpty:` method. Here's an example of how you might do this.

```
MyClass>>copyEmpty: newCapacity
   ^(super copyEmpty: newCapacity)
      varOne: self varOne;
      varTwo: self varTwo;
      yourself
```

User interface opening

When you open an application (ie, a subclass of ApplicationModel), there are several places where you can do additional processing. If you say `MyApplication open`, there are various hooks that give you the opportunity to set things up according to the needs of your application. In order, the following messages will be sent: `initialize`, `preBuildWith:`, `postBuildWith:`, and `postOpenWith:`. We'll look at each in turn, but if you decide to override them, the first thing in your method should be a message send to *super* so that the inherited method is also invoked. Here's what the fundamental interface opening method looks like. Notice that this method directly does three of the message sends just mentioned.

```
ApplicationModel>>openInterface: aSymbol
   |  spec |
   builder := UIBuilder new.
   builder source: self.
   spec := self class interfaceSpecFor: aSymbol.
   self preBuildWith: builder.
   builder add: spec.
   self postBuildWith: builder.
   builder window model: self.
   builder openWithExtent: spec window bounds extent.
   self postOpenWith: builder.
   ^builder
```

In what follows, we will use the example of a window with two action buttons and two input fields. The action button actions are *actionOne* and *actionTwo* and their IDs are *actionOneAB* and *actionTwoAB*. The input fields have aspects of *inputOne* and *inputTwo*, with IDs of *inputOneIF* and *inputTwoIF*. If you create the window and the appropriate methods, you can see that how this all works. The code can be found in the file `hooks.st`.

initialize

The initialize method is where we usually set up the various ValueHolders for the aspects, assuming we don't want to use lazy initialization. In our example we will also set the field data.

```
MyClass>>initialize
   super initialize.
   inputOne := 'Text in input field one' asValue.
   inputTwo := 'Text in input field two' asValue
```

preBuildWith: aBuilder

The `preBuildWith:` method allows you to change the way that the user interface will be built. The builder uses the spec that it is supplied with (usually #windowSpec, which is the symbol used if the `open` message is sent), but in this method you can override the description contained in the spec. For example, we want to change the way the action buttons work. Instead of invoking the specified method when the button is invoked, we want to invoke the same method but with a different parameter. We can set this up in `preBuildWith:`. We specify a block of code that should be executed when the button is pressed.

```
MyClass>>preBuildWith: aBuilder
   super preBuildWith: aBuilder.
   aBuilder actionAt: #actionOne put: [ self doAction: #one ].
   aBuilder actionAt: #actionTwo put: [ self doAction: #two ].
```

We could also set up blocks of code to build menus every time a menu button or a pop-up menu is selected. This would allow us to dynamically change the menu based on varying conditions. For example, we might have code such as the following, where `buildInputOneMenu` is invoked every time the user wants to see the menu.

```
aBuilder menuAt: #inputOneMenu put: [self buildInputOneMenu]
```

postBuildWith: aBuilder

In `postBuildWith:` we can make changes to the interface after it's built but before it is displayed. In our example, we will set the label of one button and the visibility of the other button. These actions can only be take after the window is built.

```
MyClass>>postBuildWith: aBuilder
   super postBuildWith: aBuilder.
   self invisibleButton
      ifTrue:
         [(aBuilder componentAt: #actionOneAB) labelString: 'Yes'.
         (aBuilder componentAt: #actionTwoAB) beInvisible]
      ifFalse:
         [(aBuilder componentAt: #actionOneAB) labelString: 'No'.
         (aBuilder componentAt: #actionTwoAB) beVisible]
```

postOpenWith: aBuilder

In `postOpenWith:` we can do things that require the screen to already be displayed. In our example, we specify which input field should have the focus — ie, which field the cursor should appear in, and where to position the cursor in the field.

```
MyClass>>postOpenWith: aBuilder
   super postOpenWith: aBuilder.
   self invisibleButton
      ifTrue:
         [component := aBuilder componentAt: #inputOneIF.
         component takeKeyboardFocus.
         component widget controller selectAt: self inputOne value
size + 1]
      ifFalse:
         [component := aBuilder componentAt: #inputTwoIF.
```

```
component takeKeyboardFocus.
component widget controller selectAt: 1]
```

Support methods

Here is sample code for the other methods we need to write.

```
MyClass>> invisibleButton
   "Requires an instance variable called invisibleButton"
   ^ invisibleButton isNil
      ifTrue: [invisibleButton:= Dialog confirm: 'Invisible button?']
      ifFalse: [invisibleButton]

MyClass>>doAction: aSymbol
   Transcript cr; show: 'Action ', aSymbol

MyClass>>inputOne
   ^inputOne

MyClass>>inputTwo
   ^inputTwo
```

User Interface closing

Just as there are places you can add code during the creation and opening of windows, there are places you can add code when they are closing. In particular, we will look at how you are informed that someone wants to close the window, and that the window is closing. To close a window the usual way, you send the application a `closeRequest` message (if you add a Cancel button using the Canvas Tool, you can put `closeRequest` in the Action field in the Properties Tool). The `closeRequest` message puts a close event on the event queue, as if the user had used the native window manager facilities to close the window. If you are writing a normal application subclassed off ApplicationModel, both `changeRequest` and `requestForWindowClose` messages will be sent by the window controller (an ApplicationStandardSystemController). The window will only be closed if both messages return *true*. The actual message sends are shown below, and although `requestForWindowClose` is sent to *self* (the controller), the `requestForWindowClose` method sends the same message on to the application model. Thus, the application model is sent both `changeRequest` and `requestForWindowClose`.

```
model changeRequest ifFalse: [^false].
self requestForWindowClose ifFalse: [^false].
```

Your application may need to add its own check to see whether it's okay to close the window. Typically, you will check that the user is not in the middle of editing data. If there are uncommitted changes, you'll probably prompt the user to save them. To do this checking, you can override either `changeRequest` or `requestForWindowClose`. Your method should first send the same message to its superclass, then do any application specific checking, returning *true* if it's okay to close the window, and *false* otherwise.

changeRequest

The `changeRequest` message is sent when something wants to change and is trying to find out if it's okay to change. In our case the change is quite drastic — we want to close the window. The window's controller sends

changeRequest to the application model, asking if the application model gives permission for the change (ie, the close). The inherited implementation of changeRequest sends an updateRequest message to all the dependents of the application model, asking whether they think that it's okay to change. So, changeRequest just checks with all the application model's dependents.

We can override changeRequest to add our application checks, making sure that the superclass's method is still invoked. Returning *true* now means that both the application model and its dependents agree to the change. Here's an example.

```
MyClass>>changeRequest
   super changeRequest ifFalse: [^false].
   ^self hasUncommittedChanges
      ifTrue: [self checkCloseWithUser]
      ifFalse: [true]
```

In fact, returning *true* simply says that from the perspective of the application model and its dependents, it's okay to change. The controller will then check with the keyboard processor to make sure that there are no fields needing validation. So it's possible the window will not close even though you return *true*. In VisualWorks 1.0, overriding changeRequest was the standard technique for adding your own logic to see if the window may be closed. However, since the window might remain open even though you return *true*, it's a better technique to override requestForWindowClose, which was added in VisualWorks 2.0.

requestForWindowClose

After sending changeRequest, the window controller then sends the requestForWindowClose message to its application model, which checks to see if the keyboard processor thinks it's okay to close the window. If the keyboard processor returns *true*, no other checking will be done, and the window will be closed. This is therefore a good time to do your application checking. Here's an example.

```
MyClass>>requestForWindowClose
   super requestForWindowClose ifFalse: [^false].
   ^self hasUncommittedChanges
      ifTrue: [self checkCloseWithUser]
      ifFalse: [true]
```

Overriding requestForWindowClose is now the recommended technique for adding your own checking to see if the window is allowed to close.

noticeOfWindowClose:

When a window closes, its application model will be sent the message noticeOfWindowClose:. If your application inherits this from *ApplicationModel*, the method simply returns *self*. However, you can override noticeOfWindowClose: if you want to do something specific when the window closes. For example, when I open windows from other windows, I set up a parent-child relationship. This is different from the VisualWorks master-slave relationship because in the master-slave relationship a slave can't be a master in another relationship, whereas a window can simultaneously be the child of one window and the parent of another window. When a parent window closes, it closes all its children. Here's an example of how this works, using the noticeOfWindowClose: message.

```
MyApplication>>requestForWindowClose
   ^super requestForWindowClose
      ifTrue: [self childApplicationsCanClose]
      ifFalse: [false]

MyApplication>>childApplicationsCanClose
   "We don't know if child applications will override changeRequest
   or requestForWindowClose, so we'll check both."
   | childrenThatCantClose |
   childrenThatCantClose := childApplications
      select: [:each | each changeRequest == false
                  or: [ each requestForWindowClose == false]].
   ^childrenThatCantClose isEmpty

MyApplication>>noticeOfWindowClose: aWindow
   super noticeOfWindowClose: aWindow
   self parentApplication notNil
      ifTrue: [self parentApplication removeChild: self].
   childApplications do:
      [:each | "Don't send closeRequest to subcanvases"
         each builder window == self builder window
            ifFalse: [each closeRequest]]
```

In Chapter 27, Extending the Application Framework we talk about extending the VisualWorks ApplicationModel. The code shown above extends the extensions we discuss in Chapter 27 and can be found in the file `framewrk.st`.