Chapter

21

Debugging

This chapter is not so much about debugging, as about debugging with Smalltalk. It won't necessarily help you become a better debugger, but it will help you learn what tools are available to help debug a Smalltalk application. One hint to becoming a better debugger is to explain the problem to someone else. The act of thinking the problem through clearly enough to put into words often leads you to a solution without the other person having to say anything. If you can distill out the critical factors that make this so, you can become a better debugger even when no one else is present.

Bringing up a Notifier window

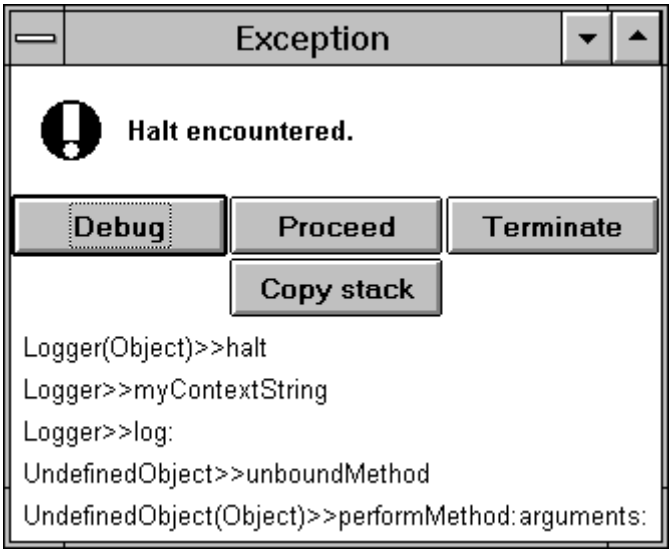

Figure 21-1.
A notifier window.

The basic mechanism for halting execution and giving you a Notifier window, as shown in Figure 21-1, is to send the `halt` message in your code. The normal way of doing this is to send it to *self*. For example,

```
self halt.
```

When the `halt` message is sent, the halt method raises a signal (*Object haltSignal*), which eventually invokes the *EmergencyHandler*. The default emergency handler places a Notifier window on the screen which gives you the option to debug the program. (Actually, it asks the class *NotifierView* to open. NotifierView opens a Debugger which gives you the option of debugging, proceeding, terminating, etc. Since the window you get when you press Debug is what is usually thought of as the Debugger, it seems convenient to refer to the initial window as a Notifier window.) If you have a string you'd like displayed in the Notifier window, send the `halt:` message instead of `halt`. For example,

```
self halt: 'My string to display'.
```

While sending one of the `halt` messages is the normal way to bring up a Debugger, you can also send `notify:` with a string parameter or `error:` with a string parameter. For example,

```
self notify: 'Count is now over 100'.
self error: 'Should not get here'.
```

Using a Debugger window

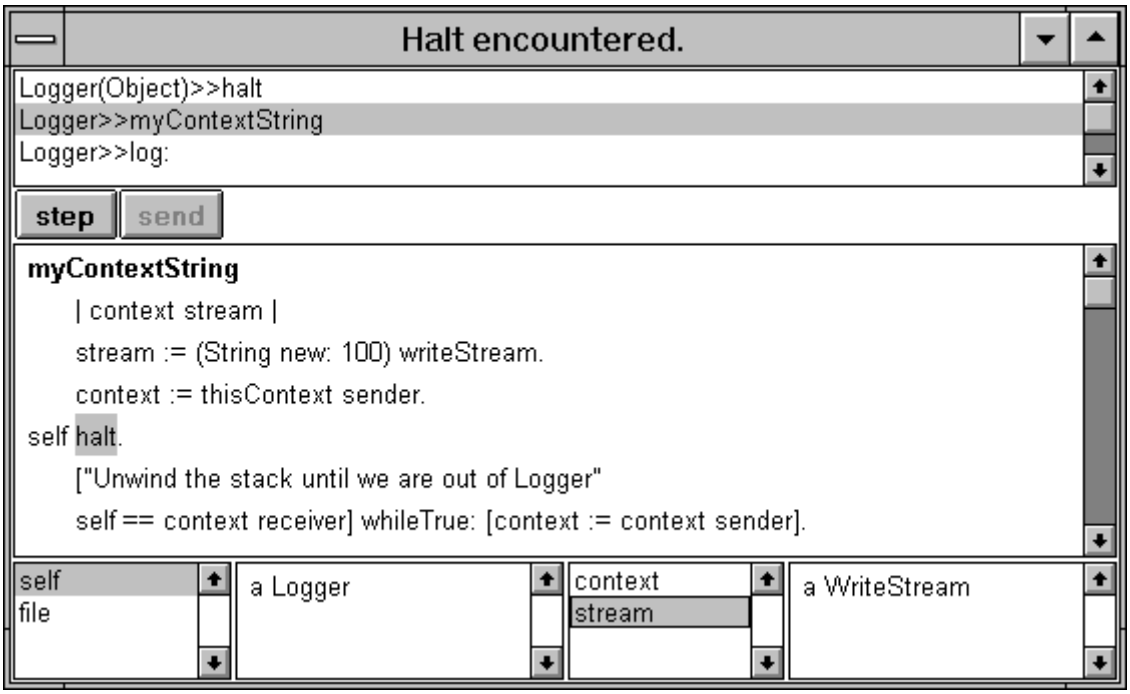

Figure 21-2. A debugger window.

From the Notifier window, you can press the Debug button to bring up a Debugger. In the top window, the Debugger shows you several lines of the stack trace. By selecting one of the lines, you can see the source code for that method. You can examine all the parameters and temporary variables for the method, and all the class and instance variables for the message receiver. If the stack trace does not show enough methods, you can show more of the stack by selecting the *more stack* mouse menu option in the upper pane.

The main window shows the code. In this window you can highlight code segments and execute them, you can modify the code and accept it, or you execute the code one message at a time. To simply execute the next

message send, press the *step* button. If you want to trace through the message send and go into the method that is executed, press the *send* button. The Debugger will stop at the first message in the new method.

If you modify the code and accept it, execution will start again at the beginning of the method if you press restart or proceed. This allows you to replay the application to try and recreate a problem (assuming that the application can be restarted). Go down the stack to a method that you can get useful debugging information from if you restart. Adding then removing a space forces recognition that a change has happened to the code and you can then accept the method. Now you can use the *step* and *send* keys to step through the code and try to understand why the problem occurred.

A useful technique for writing new code is to write most of the code in the Debugger. Write your main method as a series of message sends that invoke supporting methods to do the real work. Each supporting method will initially consist of nothing more than `self halt`. When you run your application, you will get a Debugger on the first supporting method. You can now write the real code in the Debugger, using it to help you understand the context and the appropriate message sends, and to test out possible options.

Another use for the Debugger is to step through your code, one line at a time, when you first run your application. In doing this, you will inevitably gain insights into the code. You'll see obvious mistakes that you made, and obvious ways to restructure your code. Stepping through your code in this way will almost always help you improve the code, making it less likely that you will need the Debugger to later track down bugs.

Inspectors

At the bottom of the Debugger are two Inspectors. The left one shows class and instance variables, while the right one shows parameters and temporary variables. When you click on a variable name, its value will be displayed in the right hand window of the Inspector. To open a new Inspector, you can either highlight a variable name or a code sample in the code window then choose *inspect* from the right mouse button menu, or you can select a variable name in an Inspector and choose *inspect* from the mouse menu.

(When you open an Inspector, the object you want to inspect is sent the `inspect` message. Most objects inherit `inspect` from *Object*, which opens a generic Inspector. However, some objects, such as *OrderedCollection*, *Dictionary*, and *String* override `inspect` to open a specialized Inspector. You can use `basicInspect` to open a generic Inspector.)

Modifying values in an Inspector

In the examples that follow, we will use a class called ModifyingValues, with three instance variables. The code for this example can be found in the file `debug.st`. To bring up the Debugger window, evaluate `ModifyingValues new initialize`. The initialize method is as follows.

```
ModifyingValues>>initialize
   color := #blue.
   size := 5 -> 10.
   things := OrderedCollection with: 'one' with: 'two' with: 'three'
with: 'four'.
   self halt: 'Use an inspector to modify the values'
```

If you want to replace a value in a variable, use the Inspector windows at the bottom of the Debugger. Highlight the variable, which causes the contents to be displayed on the right hand window of the Inspector.

Replace the value in the window with the new value, and choose *accept* from the mouse menu. In the following example, replace the value #blue with another value, then accept the change.

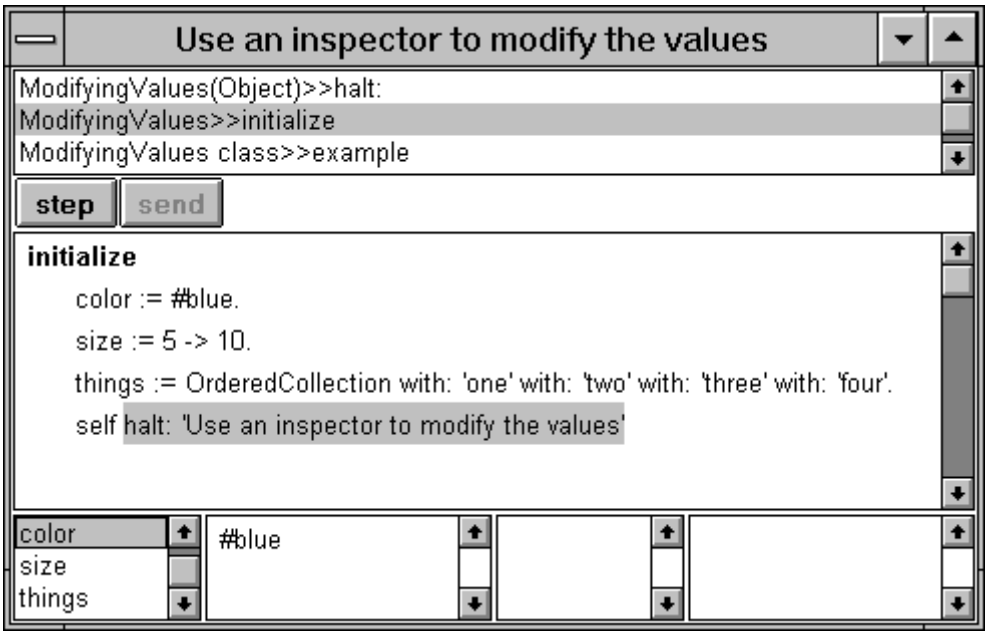

Figure 21-3. Changing a value in the Inspector.

On the other hand, if you want to replace the *contents* of an object that is being referred to by a variable, you will need to bring up a new Inspector on the variable. You can do this to modify an instance variable of an object, or to modify the contents of a collection. In this example, we want to modify the *things* collection, removing one item and adding another. If we bring up an Inspector on the collection, we get a window like the following. Now we select an item in the collection and use the mouse menu to delete it, or we can use the mouse menu to insert a new item.

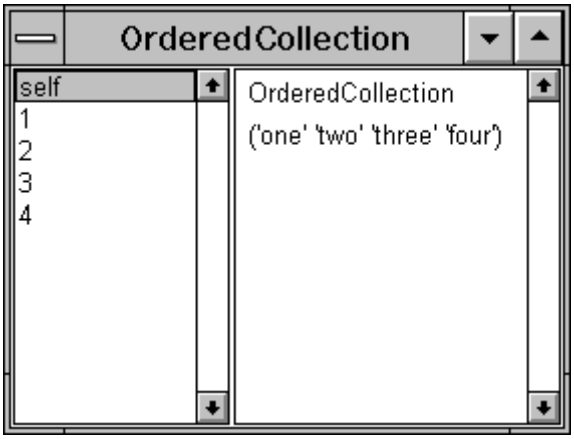

Figure 21-4.
Inspecting an OrderedCollection.

In our example, the *size* variable is an Association. If we want to replace part of the contents of *size*, we again bring up a new Inspector on the variable. We then select the variable we want replaced, change the value, and accept the new value.

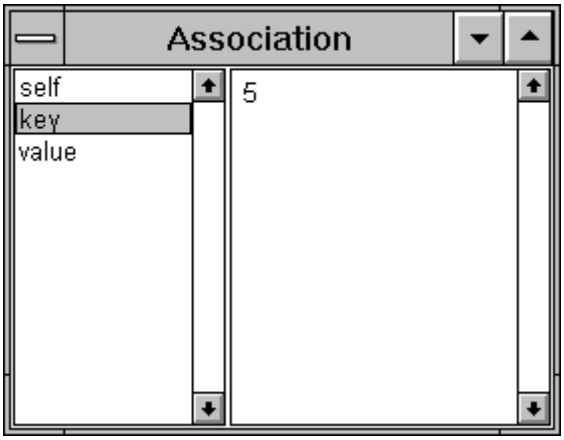

Figure 21-5.
Inspecting an Association.

Halting

If you have several occurrences of `self halt` in the code, it can be difficult to figure out where a particular Notifier window came from, especially if you are using forked processes. Sometimes you want to proceed a sequence of halts until you find a halt generated from a particular place. The Notifier windows do show some of the stack, but I find it easier to look at the text at the top rather than scan the stack. There are two ways of doing this: you can modify system methods or write your own.

Modifying halt and halt:

The easiest way to provide context information is to modify `halt` and `halt:` for *Object*. Some people recommend against modifying *Object*, but this is a change that is easy to reproduce when you get a new version of Smalltalk, and it doesn't much matter if you don't make the change. Modifying the development tools and development environment tends to be less controversial than modifying system classes that will be used in the deployed application. In both these examples we want to find the sender of the message, and we get this from `thisContext sender`.

In `halt`, modify the `errorString:` parameter to be `'Halt in ', thisContext sender printString`.

In `halt:`, modify the `errorString:` parameter to be `'Halt in ', thisContext sender printString, ' - ', aString`.

Writing your own myHalt and myHalt:

If you prefer not to modify system methods, or you have organization rules that prohibit this, then you could instead write new methods on *Object*, `myHalt` and `myHalt:`, which you would use instead of `halt` and `halt:`.

```
Object>>myHalt
   Object haltSignal
      raiseRequestWith: thisContext
      errorString: 'Halt in ' , thisContext sender printString
```

```
Object>>myHalt: aString
   Object haltSignal
      raiseRequestWith: thisContext
      errorString: 'Halt in ' , thisContext sender printString , ' -
' , aString
```

Conditional halting

There are times when you don't want to halt every time through. One option is to conditionally do the `self halt`. Alternatively, you could write a `haltIf:` method on *Object*. Here are examples of the two approaches.

```
(some condition) ifTrue: [self halt]

Object>>haltIf: aBlock
   aBlock value ifTrue:
      [Object haltSignal
         raiseRequestWith: thisContext
         errorString: 'Halt in ' , thisContext sender printString]

self haltIf: [some condition]
```

Sometimes you don't have a specific condition you want to test for. You can see things happening on the screen or on the Transcript, and after a while you want to trigger the `self halt`. You need some stimulus to tell the debug code that you are ready, and one obvious way is to hold down a key or key combination that is not pressed in normal operations. For example, we might write an `altCtrlHalt` method that only halts when the Alt and Ctrl keys are held down at the same time (you could of course write a similar `altHalt`, `ctrlHalt`, `shiftHalt`, etc.)

```
Object>>altCtrlHalt
   "Note the two mechanisms we can use for detecting if the key is
down.
   The first one is about 50% faster than the second."
   (InputState default altDown
      and: [ScheduledControllers activeController sensor ctrlDown])
         ifTrue: [Object haltSignal
            raiseRequestWith: thisContext
            errorString: 'Halt in ' , thisContext sender printString]
```

You could probably leave `altCtrlHalt` message sends in the code for long periods of time, and possibly leave them in customer code for over-the-phone debugging (assuming the stripped-down image contains the classes necessary to run the Debugger). If you don't want to halt, but would rather log something or display something to the Transcript, you can use `altCtrlDo:`.

```
Object>>altCtrlDo: aBlock
   (InputState default altDown and: [InputState default ctrlDown])
         ifTrue: [aBlock value]
```

Halting without adding a halt

A useful technique for stepping through code without having to add `self halt` to a method is to do the following. If you highlight the code and execute it, you will get a Debugger window and will be able to step into `someMethod`.

```
self halt.
someObject someMethod.
```

Writing information to a file or the Transcript

Sometimes you may want to write information about the context to a file or to a Transcript for debugging. Here's an example of a class that does some simple logging The code that follows prints a string followed by a printout of the stack, showing all the message sends that you made. It will log to either the Transcript or to a file, depending on whether the *file* variable has been set (the setting method is not shown here). This code can be found in the file `debug.st` (to run it, evaluate `LoggerExample new run` in the Transcript after filing it in).

```
Logger>>contextString
   | context stream |
   stream := (String new: 100) writeStream.
   context := thisContext sender.
   ["Unwind the stack until we are out of Logger"
   self == context receiver] whileTrue: [context := context sender].
   "Unwind the stack as far as possible, printing out as we go"
   [context notNil and: [context receiver notNil]]
      whileTrue:
         [stream cr; print: context.
         context := context sender].
   ^stream contents

Logger>>log: aString
   | contextString |
   contextString := self myContextString.
   Object errorSignal
      handle: [:ex |file := nil.
            ex restartDo: [self log: aString]]
      do: [self myWrite: self myWriteStream string: aString context:
contextString]

Logger>>myWrite: aStream string: aString context: aContextString
   [aStream cr; cr; nextPutAll: aString; nextPutAll: aContextString;
flush]
      valueNowOrOnUnwindDo:
         [aStream == Transcript ifFalse: [aStream close]]

Logger>>myWriteStream
   ^file notNil
      ifTrue: [file appendStream]
      ifFalse: [Transcript]
```

You can use the class *NotifierView* to give you stack information by doing the following message send, which returns a string containing a stack dump. The second parameter specifies the depth, or number of stack references to return. If the stack is not as deep as the depth you specify, the string contains the full stack. Note the that message is in the *private* protocol, which means it might be changed or removed.

```
NotifierView shortStackFor: context ofSize: 99
```

For more information on logging debug information, see the article *A Trace Logger*, by Alec Sharp and Dave Farmer, in The Smalltalk Report, Nov/Dec 1994.

Audible and Visible information

Sometimes you may simply want to know when something is happening. If you are on a system that can make a bell sound, you can put the following code into methods to indicate that they are being executed:

```
Screen default ringBell.
```

If you don't want to ring a bell, you can flash a widget. Choose a widget (for example, a widget with the name saveAB), then send it the `flash` message. For example

```
(self builder componentAt: #saveAB) flash.
```

Monitoring activity of objects

If you have an object that doesn't seem to be behaving itself but you can't figure out why, you can create an object to watch it and report on its behavior. To do this, create a *Watcher* class as follows. Notice that the superclass is *nil*. You'll get a warning message about a *nil* superclass when you accept the class definition for *Watcher*, and also if you fileIn the code. When you get the warning window, press the Proceed button.

```
nil subclass: #Watcher
   instanceVariableNames: 'watchedObject '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Debugging'

Watcher class>>on: anObject
   ^self new initializeWatcher: anObject

Watcher>>initializeWatcher: anObject
   watchedObject := anObject

Watcher>>doesNotUnderstand: aMessage
   | result |
   Transcript
      cr; print: thisContext sender; nextPutAll: ' sent ';
      print: watchedObject class; nextPutAll: '>>'; nextPutAll:
aMessage selector asString.
   aMessage arguments size > 0 ifTrue:
      [Transcript print: aMessage arguments].
   result := watchedObject
      perform: aMessage selector
      withArguments: aMessage arguments.
   Transcript nextPutAll: ' ^'; print: result; flush.
   ^result
```

When you've defined the class and its methods, file out the code, remove the class, then file the code back in. This will create any additional needed methods. (In particular, it creates the method `class`. Since *Watcher* is subclassed off *nil*, it has some special needs.) To use the Watcher, instead of creating your object in the code, create a Watcher on the object by doing something like

```
myObject := Watcher on: (MyClass new).
```

In your code, all the messages that are being sent to myObject will actually go to the Watcher, which keeps the "real" object in an instance variable. Being subclassed off *nil*, the Watcher doesn't understand many

messages, so messages sent to it will end up invoking the `doesNotUnderstand:` method. In that method we log the message and its arguments, and who sent the message. We then pass the message on to its correct destination, log the return value, then return the return value. It would be easy enough to modify this code if you wanted to log to a file rather than to the Transcript. Here's an example, written on the class side of Watcher, followed by the results.

```
Watcher class>>example
   "self example"
   | array |
   array := self on: (Array new: 3).
   array at: 2 put: 'hello'.
   array at: 2.
   array printString.
   array size

Watcher class>>example sent Array>>at:put:#(2 'hello') ^'hello'
Watcher class>>example sent Array>>at:#(2) ^'hello'
Watcher class>>example sent Array>>printString ^'#(nil ''hello''
nil)'
Watcher class>>example sent Array>>size ^3
```

A brief aside

As an aside, the technique of overriding `doesNotUnderstand:` makes it possible to add instance specific behavior to instances of a class. Suppose MyClass has an instance variable called *methods* (with accessors) which contains an IdentityDictionary. The Dictionary could contain associations of methods and blocks of code. MyClass would override `doesNotUnderstand:` as follows.

```
MyClass>> doesNotUnderstand: aMessage
   | codeBlock |
   codeBlock := self methods at: aMessage selector ifAbsent: [^nil].
   codeBlock valueWithArguments: aMessage arguments
```

Here's an example of how it might be used. When you execute this code, 'YES' is printed to the Transcript.

```
instance := MyClass new.
dictionary := IdentityDictionary new.
dictionary at: #yes put: [Transcript cr; show: 'YES'].
instance methods: dictionary.
instance yes.
```

This concept can be extended further to give *roles* to objects. To use an everyday example, most people have several roles during the day. For example, a woman may at different times play the role of wife, mother, employee, boss, shopper, and chauffeur. Each role has different behaviors. Smalltalk does not have multiple inheritence, but we can emulate the concept with roles. Suppose we have a RolePlayer with an instance variable of *role*. Instead of keeping a Dictionary of methods and blocks of code, we would create a Role object and pass the Role object to the RolePlayer. Then, if a message is not understood by the RolePlayer, it would pass it on to the Role. For example,

```
RolePlayer>>doesNotUnderstand: aMessage
   role notNil ifTrue:
      [^role perform: aMessage selector withArguments: aMessage
arguments]
```

```
ChauffeurRole>>pickUpKids
   Transcript cr; show: 'I am picking up the children'
```

Then, if we execute the following code, we see the message 'I am picking up the kids' printed to the Transcript.

```
person := RolePlayer new.
person role: ChauffeurRole new.
person pickUpKids.
```

You can find the code for this example in the file `role.st`. The topic of roles is quite a bit more involved than described here and is beyond the scope of this book. However, what we've just described may give you ideas for solutions to your own application problems.

Objects not being garbage collected

During development, the issue of objects not being garbage collected doesn't usually arise because your focus is getting the functionality written. However, as you approach shipping time, you often find that the memory used by your image keeps growing as the application is used. This usually means that there is a memory leak because you have objects that are not being garbage collected. When an object is not garbage collected, it means that it is still being referenced by at least one other object. And this probably means that the other object is not being garbage collected either.

If you think you have objects hanging around longer than they should, start by doing a garbage collection. You can do this from the File menu in the Launcher, or by doing one of the garbage collection operations in the `collecting garbage` class side protocol of *ObjectMemory*. Try one of the verbose operations, which give you information on the garbage collection done. Once you've cleaned up the garbage, you can look at all instances of the class you think has unreclaimed instances by inspecting

```
SomeClassName allInstances.
```

If there are several instances, by inspecting the instance variables, you may be able to find one that you know should have disappeared. Otherwise, you'll have to select one at random. You can now look at the objects that reference this instance by inspecting `self allOwners` in the Inspector window. This is the classic approach, but it's not easy to use this technique to follow through the owners. A better approach is to inspect the following in the Inspector window.

```
self allOwnersWeakly: true
```

Sending `allOwnersWeakly: true` to an object returns a WeakArray of references rather than an Array, which means that the references can be garbage collected. Using `allOwnersWeakly:` usually gives you a smaller list of owners to look at. Below is some code that you can add to *Inspector* to inspect owners of an object, filtering out most of the uninteresting owners. It is based on a method written by Jan Steinman and documented by Alan Knight in The Smalltalk Report, May 1994. It appears to work, but I don't use it enough to guarantee anything!

```
Inspector>>inspectOwners
```

```
   ((self fieldValue allOwnersWeakly: true) reject:
      [:each | self shouldReject: each]) inspect

Inspector>>shouldReject: anObject
   "The WeakArray often gives an owner of 0"
   anObject == 0 ifTrue: [^true].
   "We don't want to see the object we are inspecting"
   anObject == object ifTrue: [^true].
   "We don't want to see this inspector"
   anObject == self ifTrue: [^true].
   "We don't want to see the methods that got us here"
   (anObject class == MethodContext
      and: [anObject selector == #allOwners
      or: [anObject selector == #allOwnersWeakly:]]) ifTrue: [^true].
   "We don't want to see the stack array that contains the object"
   (anObject class == Array
      and: [anObject size == 12
      and: [(anObject at: 3) == self fieldValue]]) ifTrue: [^true].
   "We want to see anything that's left after this filter"
   ^false
```

The Advanced Tools contains a class, *ReferencePathCollector*, whose purpose is to help find objects that are not being garbage collected. The class comment includes the following: "My purpose is to find what is hanging onto objects that can't be garbage collected, by finding reference paths to a given object." We can add another item to the inspect menu that uses the ReferencePathCollector. When you use it, be prepared for a long path; it can be several hundred references long.

```
Inspector>>inspectPaths
   (ReferencePathCollector allReferencePathsTo: self fieldValue)
inspect
```

Once you've added these methods, you'll need to go to the `fieldMenu` method of Inspector and add the methods `inspectOwners` and `inspectPaths` to the menu. The labels string should now look something like *'inspect\owners\paths' withCRs*. Accept the method then evaluate `Inspector flushMenus` to make the change take effect. Here's a few lines of code from the modified `fieldMenu`.

```
ListMenu == nil ifTrue:
   [ListMenu := Menu
      labels: 'inspect\owners\paths' withCRs
      values: #(inspectField inspectOwners inspectPaths)].
```

For additional information on objects not being reclaimed, you might look at the article *Taking Out The Garbage*, by Derek Williams, in the January 1996 issue of The Smalltalk Report.

Public Domain debugging software

There is a debugging enhancement available from the Smalltalk Archives, written by Bob Hinkle. Called *Breakpoint*, its great virtue is that it allows you to add and remove breakpoints without affecting the change set and change log. You can add either an absolute breakpoint, which always causes a halt when encountered, or a conditional breakpoint, which only causes a halt when the associated condition evaluates to *true*. A companion package, *Lightweight*, also written by Hinkle, allows you to add breakpoints to individual objects, rather than all objects of that class. Ie, you can modify the behavior of just the one object without affecting the behavior of any

other object. There is more information on how to get files from the Smalltalk Archives in Chapter 35, Public Domain Code and Information.

Commercial debugging software

The Smalltalk Professional Debug Package

Also available for use with Objectworks\Smalltalk and VisualWorks is the *Smalltalk Professional Debug Package*. As with *Breakpoint*, adding a breakpoint does not affect the change set or the change log. However, where *Breakpoint* uses the compiler to add breakpoints, the *Smalltalk Professional Debug Package* modifies the compiled code directly. This provides the capability for some nice features such as the ability to add another breakpoint to a method without having to restart the method, having the breakpoints disappear when the method exits, and allowing skip-to-caret into and out of blocks. You can also use the package to add debugging code that does not cause a halt; for example to just collect data.

At the time of writing, the software cost is $89.00 plus shipping and handling ($149 for the ENVY version). To order or get additional information, you can send e-mail to traymond@pcix.com, or call Terry Raymond at (401) 846-6573. He also has a web site at http://www2.pcix.com/~traymond/ where you can get more information.

Object Explorer

Another package that might help you debug an application is *Object Explorer* from First Class Software. This software diagrams the inter-relationships between objects, which can be quite a help when debugging a complex application that you are not familiar with. There is a review of Object Explorer by Steven Bilow in the June 1994 issue of The Journal of Object-Oriented Programming, which speaks very highly of the product—"It is a profound debugging tool and a great design aid."

At the time of writing, the software cost is $499. You can reach First Class Software at P.O. Box 226, Boulder Creek, CA 95006-0226, at (408) 338-4649, or by e-mail to Compuserve address 70761.1216@compuserve.com.