Debugging Objects[†]

Bob Hinkle[*], Vicki Jones, and Ralph E. Johnson

Department of Computer Science
University of Illinois at Urbana-Champaign
1308 W. Springfield Ave., Urbana, IL 61801

r-hinkle@uiuc.edu
{vjones, johnson}@cs.uiuc.edu

As the premier object-oriented programming language, Smalltalk should give programmers easy access to objects. However, during debugging it can sometimes be very difficult to get your hands on a particular object. For example, suppose you're developing a program that stores some objects in an OrderedCollection, but when it tries to retrieve them later, some are missing. You might like to add debugging code to OrderedCollection methods such as add: and remove: to detect when objects are taken out of the OrderedCollection, but any changes would affect every OrderedCollection in the system, bringing your image to a crashing halt. This article will show how to solve this and similar problems by letting you modify code and add breakpoints that affect only one specific object, rather than all objects in a given class. Our solution relies on the use of a new kind of class and on some small but powerful variations on CompiledMethods and Compilers. Besides being useful in their own right, we feel these extensions again illustrate (as in our previous article, [1, 2]) how powerful the reflective features of Smalltalk are, as they allow the programmer to adapt and extend the environment to suit his or her needs. The solution described is specific to Smalltalk-80, since it relies on Smalltalk-80's architecture for classes, metaclasses, the compiler, and compiled methods, and on the complete availability of source code for these system elements. As a result, our extensions may not apply to Smalltalk V environments, though something similar may be possible.

Lightweight Classes

The first step to debugging objects is to be able to modify methods on a per-object basis. In Smalltalk, methods for an object are defined in that object's class and are stored in the class' method dictionary. To change a method for one particular object requires that the object have its own private class. We will give an object that we want to debug its own class by inserting a new class between the object and its real class. We could create a (perhaps temporary or anonymous) instance of class Class for this purpose, but that's a little heavy-

---

[†] Source code for the object debugging package is available by anonymous ftp from st.cs.uiuc.edu. Look for the file ObjectDebugging.st in pub/st80_r41.

[*] Supported by a fellowship from the Fannie and John Hertz Foundation.

handed: instances of Class have many instance variables and a lot of behavior which aren't needed for our purposes. For example, Class adds variables and functionality to define new class and pool variables. In addition, Class inherits from ClassDescription variables and code to support adding new instance variables and class organizations. All of this is unnecessary for a lightweight class, so we defined LightweightClass to be a subclass of Behavior. Behavior is the superclass of ClassDescription, and it defines the code needed for the interpreter to do method lookup. (For more information on the roles of Behavior, Class, and ClassDescription, refer to the "Protocol for Classes" chapter of [3].) Since Behavior is a simpler starting point, instances of LightweightClass will be smaller than instances of Class and will require less memory and time to allocate, initialize, and finalize. That makes it easier and less expensive to create lightweight classes on the fly to modify, even if only temporarily, some object's behavior.

Before explaining LightweightClass in detail, it's helpful to review the way things work normally in Smalltalk. When an object is sent a message, the system tries to find a method corresponding to the message's selector in the method dictionary of the object's class. If no such method exists, the system will look in that class' superclass, and so on up the chain of superclasses until a method is found or the end of the chain is reached. Furthermore, when a method is added to a class or changed, the new code is compiled by an instance of the class' compilerClass (which by default in the system is SmalltalkCompiler). The result of compiling is an instance of CompiledMethod, which will be stored in the class' method dictionary with its selector as its key. The source code for the method is not stored directly in the CompiledMethod, but instead is written into the change log, and the CompiledMethod is given a pointer to its file and offset.

Our implementation of lightweight classes changes this normal scenario in three ways. The first and most important change inserts a LightweightClass in between an object and its real class (or what we will call original class, since it was the class by which the object was originally created), with the object's class being changed to the LightweightClass, and the LightweightClass' superclass set to the object's original class. In this way, any message sent to the object will first be looked up in the LightweightClass' method dictionary. If a method is found there it will be used to respond to the message, and it will be unique to that particular object. Otherwise message look-up will continue to the LightweightClass' superclass—the object's original class—and hence will proceed as usual for objects of that class. Figure 1 illustrates this relationship between an object, its original class, and its lightweight class.

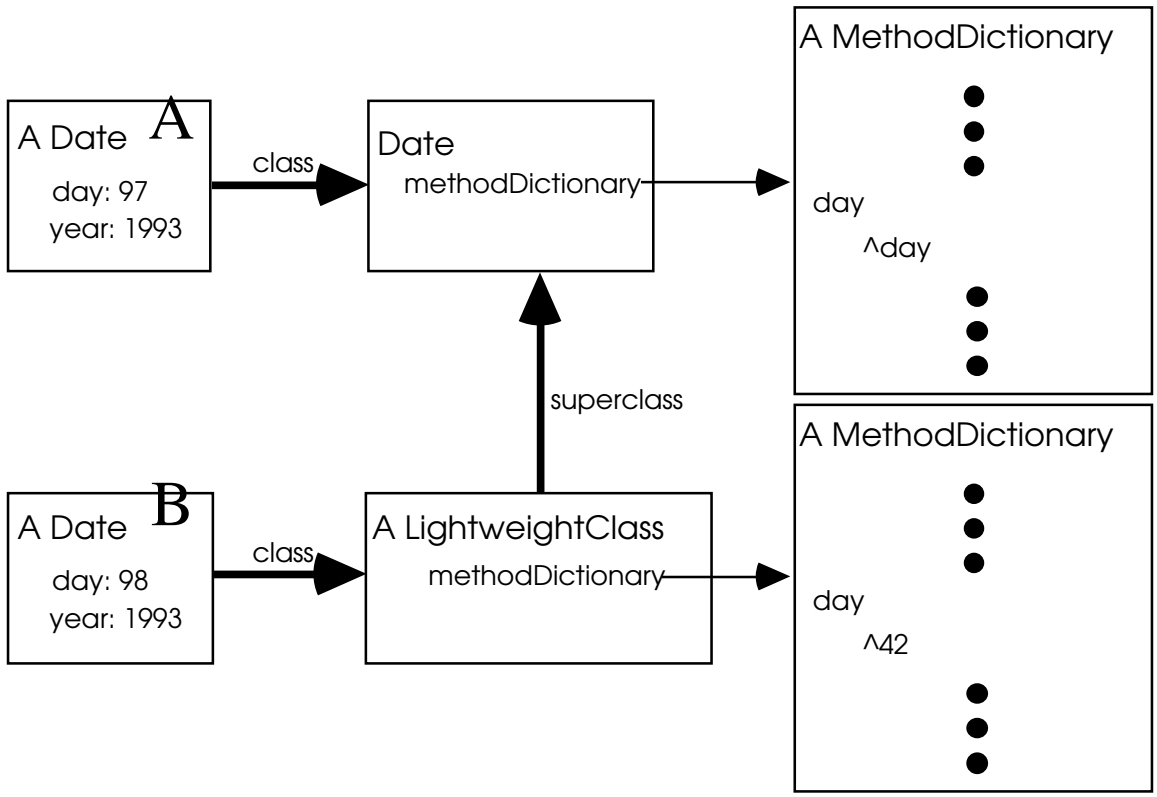

Figure 1: When the day message is sent to the object marked A, a corresponding method is searched for starting in Date, the object's class. This method returns the value of the day instance variable, which (for A) is 97. However, when the day message is sent to object B, message lookup begins in its class, which is an instance of LightweightClass. The method in the lightweight class' method dictionary is defined to return 42. Thus, object B behaves differently from A and all other instances of Date.

The two other changes pertain to source code management. The code for methods in lightweight classes can't be stored in the change log, since the lightweight class isn't named in the system dictionary, and it has no category or protocols like normal classes. (And in any case the lightweight class may be an entirely dynamic object which is created while running a program but which does not persist from one programming session to the next, so that storing code for it in the change log would make no sense.) So instead we store the code directly with the method it produces, which required us to create a new kind of compiled method, CompiledMethodWithSource. Finally, to produce these kinds of compiled methods we exploited the pluggability of the compiler and created a new subclass of SmalltalkCompiler. We'll describe these two changes after first looking at LightweightClass in detail.

As a subclass of Behavior, LightweightClass adds only one instance variable, name, which is convenient for telling lightweight classes apart. In addition to accessor methods for this variable, LightweightClass defines three other methods of interest: initializeWithSuper:, which initializes a new lightweight class; compile:notifying:ifFail:, which adds a new method to a lightweight class; and compilerClass, which defines the kind of compiler to use for methods in a lightweight class.

A new lightweight class is normally created by sending becomeLightweight to an object. This method is defined in Object as follows:

```
becomeLightweight
    | lightweightClass |
    self lightweightClass isNil
        ifTrue: [
            lightweightClass :=
                LightweightClass newWithSuper: self class.
            self changeClassToThatOf: lightweightClass basicNew]
```

If the receiver of this message already has a lightweight class, nothing more is done. Otherwise, newWithSuper: is sent to create a new lightweight class whose superclass will be the receiver object's original class. The message changeClassToThatOf: is then sent to the receiver to insert the lightweight class before the object's original class. Because some objects (notably immutable objects like SmallIntegers, Characters, true, and false) can't have their class changed, becomeLightweight can't be sent to them, but it can be sent to all others.

The newWithSuper: method creates a new lightweight class and then sends it the initializeWithSuper: message, where the parameter is the object's original class. This initialization method gives a default name to the lightweight class, creates a new method dictionary for it, and sets its superclass to be the class passed in, so that any messages not found in the lightweight class' method dictionary will be looked up in the object's original class.

The solution described in the preceding paragraphs makes sure that messages sent to a lightweight object are first looked up in the object's lightweight class as desired. However, class messages will not work correctly as the solution has been presented so far. For example, if aDay is a lightweight instance of Date, sending "aDay class nameOfDay: 1" should be the same as sending "Date nameOfDay: 1," but aDay's class is an instance of LightweightClass, so "aDay class nameOfDay: 1" will try (and fail) to find a method for the message nameOfDay: defined for LightweightClass. This problem exists because classes have several roles, including roles as method repositories and as repositories for shared information (in this case, the names of the days of the week). We want the lightweight class to play the first role and the object's original class to play the second, but Smalltalk expects one entity to play both roles. (Alan Borning summarizes the various roles of class and suggests an alternative approach in [4].) Our solution to this problem is to separate out the role of method repository, which we did by creating a new method for all objects called dispatchingClass. The definition of dispatchingClass in Object is the same as that of class—it uses a primitive to directly access the object's class from the object's memory structure. When an object is made lightweight, its lightweight class is stored in the memory structure and thus returned as the value of dispatchingClass. In addition, LightweightClass overrides the class method to be

```
class
    ^self dispatchingClass superClass
```

This will return the object's original class, as desired, since newWithSuper: installed the original class was installed as the lightweight class' superclass.

The LightweightClass method compile:notifying:ifFail: is needed when a method is defined in a lightweight class, and is implemented as:

```
compile: code notifying: requestor ifFail: failBlock
    "Compile the argument, code, as source code in the context of the
    receiver and install the result in the receiver's method dictionary. The
    argument requestor is to be notified if an error occurs. The argument
    code is either a string or an object that converts to a string or a
    PositionableStream on an object that converts to a string. This
```

```
method *does* save the source code. Evaluate the failBlock if the
compilation does not succeed."

| methodNode selector save method oldMethod |
save := code asString copy.
methodNode := self compilerClass new
                    compile: code
                    in: self
                    notifying: requestor
                    ifFail: failBlock.
selector := methodNode selector.
method := methodNode generate.
method sourceCode: save.
oldMethod := self compiledMethodAt: selector ifAbsent: [nil].
(oldMethod notNil and: [oldMethod isBreakpoint])
    ifTrue: [oldMethod client: method]
    ifFalse: [self addSelector: selector withMethod: method].
^selector
```

There are two major differences between this method and the compile:notifying:ifFail: method as defined in Behavior. First, this method saves the source code that was passed in and passes it along (using the sourceCode: message) to the CompiledMethodWithSource that's generated from the message send "methodNode generate". Also, the code checks to see whether the method being compiled used to have a breakpoint and if so preserves the breakpoint in the method dictionary. (This logic will be explained in detail in the next section.)

The final LightweightClass method is compilerClass, which simply returns a new class, LightweightCompiler, to be used when compiling lightweight class methods. Creating a new compiler class sounds over-ambitious, but it's actually quite simple, since the new class has only one method, newCodeStream—the rest are inherited straight from SmalltalkCompiler. This method is used to create a new CodeStream for use by the compiler. Since CodeStream generates CompiledMethods by default, we changed it to be parameterized by the kind of method generated, and so LightweightCompiler implements newCodeStream simply by returning a CodeStream that will generate instances of CompiledMethodWithSource. The implementation of CompiledMethodWithSource is just as simple; we changed three methods so that the sourceCode instance variable is interpreted as a source string (rather than a pointer to a file and offset), and the rest of its functionality is inherited from CompiledMethod.

With these few changes we now have an easy way to change the behavior of individual objects. We still need a good interface for doing that, though, and we'll describe our approach for that after first looking at breakpoints.

Breakpoints

One of the typical things a programmer wants to do while debugging objects (and often in other debugging, as well) is to add "self halt" to a method—effectively adding a breakpoint. As it turns out, there's a simple way to add an initial breakpoint using the same technique that we used above with LightweightCompiler and CompiledMethodWithSource; we'll simply create a new class of compiled method, BreakpointMethod, and a compiler for generating instances of it. This variety of breakpoint has three advantages over the "self halt" version: they're easier to add and remove, since it's done by menu rather than by typing; they don't affect the various change mechanisms, so the change set and change log don't include trivial changes for adding (and presumably later removing) a halt in a method;

and they're invisible in source code, so a programmer browsing or debugging a breakpointed method will see only the normally defined code—the breakpoint is invisible. The one disadvantage of our technique is that you can only halt at the start of a method, though our design may be adaptable to cover breakpoints throughout a method's body.

BreakpointMethod is a subclass of CompiledMethod with one instance variable, clientMethod. In addition, we added a new instance variable, agent, to CompiledMethod. When a breakpoint is set on an existing CompiledMethod, a new BreakpointMethod is created, and these two instance variables are changed so that the BreakpointMethod's clientMethod is the CompiledMethod, and the CompiledMethod's agent is the BreakpointMethod. The body of a BreakpointMethod is always the same: it's the expression "Notifier handleBreakpoint". Thus, when a BreakpointMethod is executed, this expression is evaluated, and Notifier responds by updating its stack, replacing the BreakpointMethod with its client—the original CompiledMethod—and opening a Debugger with that method in the top context. In this way, the BreakpointMethod itself is invisible in the debugging process, since it is removed from the execution stack before the debugger opens. In addition, BreakpointMethods implement the getSource message by returning their client's source, and so breakpointed methods can be browsed directly.

The new variable agent is needed to make CompiledMethods with breakpoints print out well. Every CompiledMethod has an instance variable called mclass, which refers to the class in whose method dictionary the CompiledMethod should be found. When CompiledMethods print themselves out, they look in their mclass to make sure they really are defined there—and if not, they will print out as an unboundMethod. Since BreakpointMethods replace their client in the method dictionary, all breakpointed methods would print out as unboundMethods, which is confusing and aesthetically unpleasing. We solved this problem by adding agent: now, when a CompiledMethod prints out, it checks to make sure that its agent is defined by its mclass, and if so it prints out normally. Most CompiledMethods are their own agents, but breakpointed methods will have their agent set to the BreakpointMethod that's representing them, and so they'll print out correctly. Figure 2 illustrates this relationship between CompiledMethods and the BreakpointMethods that represent them.

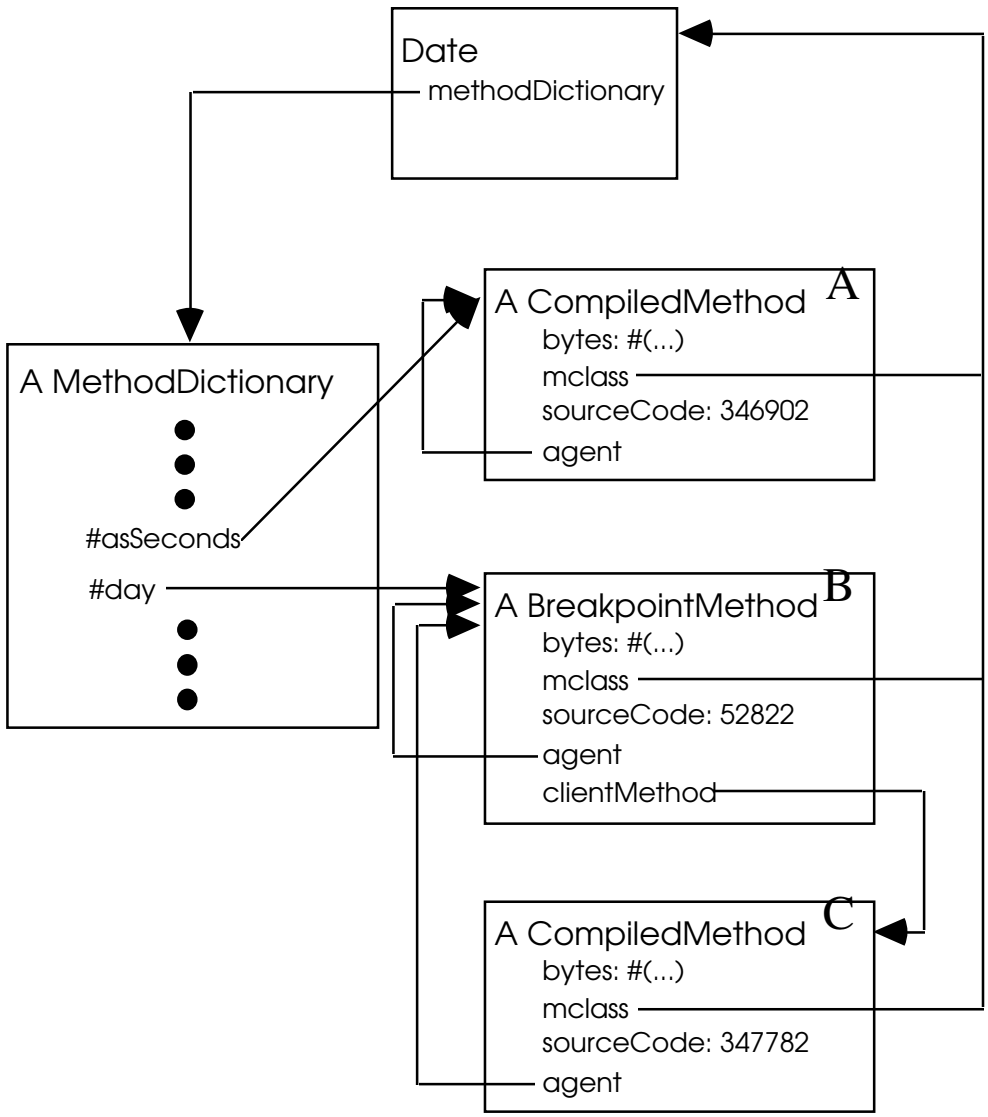

Figure 2: The asSeconds method for Date—the CompiledMethod marked A—is a normal method. Its mclass is Date, it is its own agent, and it is referred to directly by Date's method dictionary. However, a breakpoint has been placed on the day method for Date. The #day entry in Date's method dictionary refers to the BreakpointMethod B, whose clientMethod is the CompiledMethod C. CompiledMethod C, in turn, refers to BreakpointMethod B as its agent. This way, even though CompiledMethod C is not referenced by Date's method dictionary, its agent—BreakpointMethod B—is, so CompiledMethod C will print as a well-defined method rather than as an unbound one.

We added breakpoints to the system by creating three new methods in Behavior, thus making breakpoints in all kinds of classes, including instances of both Class and LightweightClass. The first method, isBreakpointAt:, tells whether the specified method in the Behavior has a breakpoint set or not. The second, breakpointCompilerClass, returns BreakpointCompiler, which is the compiler used for all classes to create new breakpointed

methods. The third method, setBreakpointAt:, is the main one and is used to set or remove a breakpoint. It's implemented as:

```
setBreakpointAt: aSelector
    | c m |
    c := self whichClassIncludesSelector: aSelector.
    c isNil ifTrue: [^self].
    m := c compiledMethodAt: aSelector.
    self == c
        ifTrue: [
            m isBreakpoint
                ifTrue: [m client mclass == self
                    ifTrue: [self addSelector: aSelector withMethod: m client]
                    ifFalse: [self removeSelector: aSelector]]
                ifFalse: [self addSelector: aSelector withMethod:
                    (BreakpointMethod on: m
                        selector: aSelector
                        inClass: self)]]
        ifFalse: [
            m isBreakpoint ifTrue: [m := m client].
            self addSelector: aSelector withMethod:
                (BreakpointMethod on: m selector: aSelector inClass: self)]
```

If the receiver Behavior is the class that defines the method corresponding to the parameter selector and if the method is already breakpointed, the code removes the breakpoint by testing whether the BreakpointMethod's client is defined in the receiver or not. If it is, the BreakpointMethod is replaced by its client in the receiver's method dictionary; but if it isn't, the BreakpointMethod is simply removed from the receiver's method dictionary (thus leaving the client in whatever other method dictionary it resides). If the method isn't breakpointed, the code creates a new BreakpointMethod for it and adds it to the receiver's method dictionary. Finally, if the method corresponding to aSelector isn't defined in the receiver, a new BreakpointMethod is created and installed in the receiver's method dictionary.

As with lightweight classes, we need a new compiler class, BreakpointCompiler, to implement breakpoints. Once again, though, this class is almost trivial, since it only needs to define newCodeStream to return a CodeStream that creates BreakpointMethods.

Putting Things Together

To exploit the functionality provided by LightweightClass and BreakpointMethod, we adapted the interface to make object debugging as simple as possible. This required changing the existing Browsers, adding a menu option to Inspectors, and creating a new Browser specifically for lightweight classes.

The existing Browsers were changed by adding a breakpoint option to the menu in the selector view. Choosing this option will either set a breakpoint on the selected method or, if the method's already breakpointed, remove the breakpoint, so that the option acts like a toggle switch. Furthermore, the selector view allows method selectors to be formatted, and we use a preceding asterisk to quickly distinguish methods with breakpoints.

In addition, all Inspectors now have a new menu option called browseLightweight. Choosing this option will create a new lightweight class for the selected object and open a LightweightClassBrowser to examine and modify methods for that particular object.

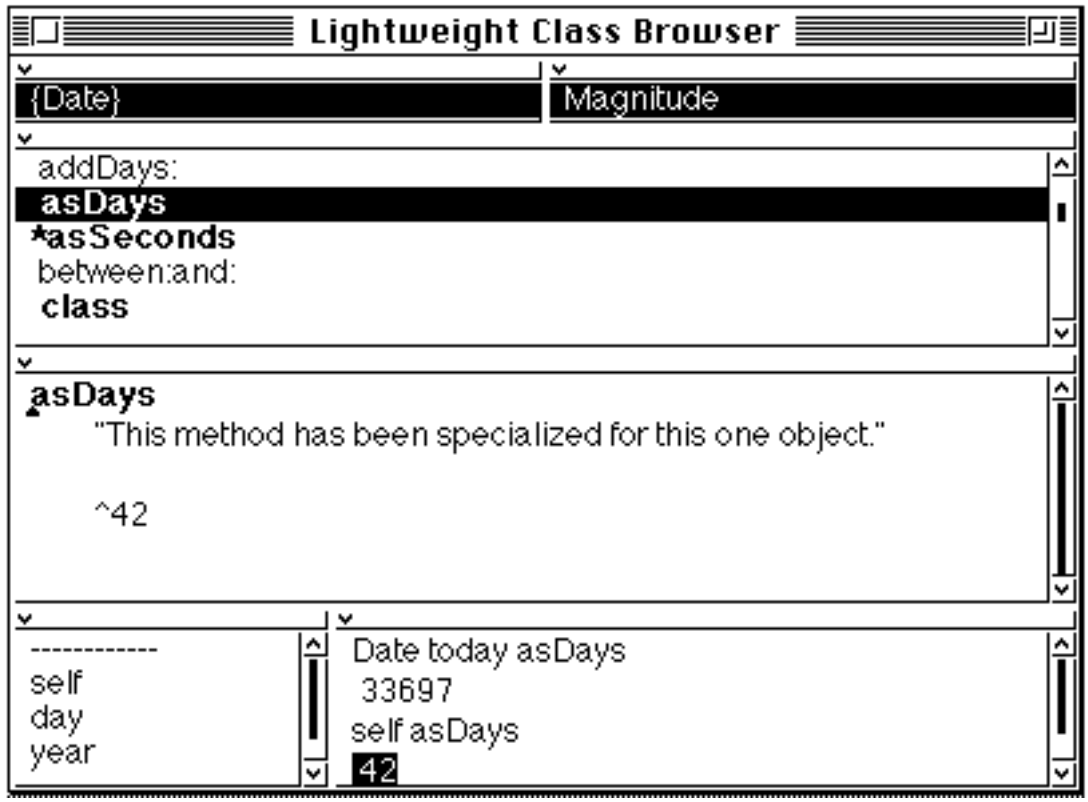

Figure 3: The Lightweight Class Browser

LightweightClassBrowser is a subclass of Browser for looking at lightweight classes. As shown in Figure 3, the LightweightClassBrowser has six subviews. The first two views allow you to decide what methods you'll see: you can either see only methods defined in the lightweight class, or all methods up to some specified superclass. The upper right view shows which class you're listing methods up to, while the upper left view shows which class the selected method is actually defined in. This option makes it easy to view a superclass method and then make changes to save in the lightweight class. The third view lists all selectors from the lightweight class up to the class chosen in the upper right view. These selectors are formatted so that all breakpointed methods are marked with an asterisk, and so that all methods actually defined in the lightweight class (as opposed to one of its superclasses) are printed in bold. The fourth view is a TextView on the code of the currently selected method. Finally, the last two views belong to an Inspector on the object whose lightweight class is being browsed.

This interface makes it easy to imagine how the debugging session mentioned in the introduction would proceed. Once you've decided there's a problem with one of your OrderedCollections, you can use a Browser to put a breakpoint on the method where the OrderedCollection is created. When that method is executed, a Debugger will pop up. The Debugger lets you inspect the OrderedCollection and choose the browseLightweight option to create a lightweight class for it. The LightweightClassBrowser lets you put breakpoints on the add: and remove: methods. After you "proceed" from the Debugger, you'll be able to watch as that one OrderedCollection is modified, and you can find out when objects are added to it and when they're removed. With that information, you'll be well on your way toward solving the problem.

These changes significantly improve debugging in the Smalltalk environment. Though breakpoints are convenient, it's the functionality of lightweight classes that makes

the key difference, as they allow you to monitor or alter the behavior of particular objects without affecting the rest of your system. The changes described here, while not complex, are remarkable in one sense, because they rely on our ability to modify parts of the Smalltalk system that in some languages would be internal and unavailable to programmers. The fact that classes are first-class objects—which is to say, classes are accessible to and modifiable by the programmer—allowed us to introduce a new kind of class and to replace an object's class on the fly during execution. Similarly, we were able to create two subclasses of CompiledMethod, and make an important change to that class itself, only because compiled methods are first-class. Finally, Smalltalk's representation of the Compiler itself, and its good design for pluggability, allowed us to create two simple subclasses by defining only one method each. The combination of the ease of making these changes with the significant benefits they provide is a good argument for the desirability of this level of reflection in a programming system. In our next article we plan to explore one level deeper into Smalltalk's reflectiveness by changing the compiler and the interpreter to introduce active variables and watchpoints.

References

[1] Bob Hinkle and Ralph E. Johnson. *Taking Exception to Smalltalk, Part 1.* The Smalltalk Report, Vol. 2, Number 3, November/December 1992.

[2] Bob Hinkle and Ralph E. Johnson. *Taking Exception to Smalltalk, Part 2.* The Smalltalk Report, Vol. 2, Number 4, January 1993.

[3] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley Publishing Company, 1983.

[4] A. H. Borning. *Classes Versus Prototypes in Object-Oriented Languages.* Proceedings of the ACM/IEEE Fall Joint Computer Conference. Dallas, TX, November 1986, pp. 36-40.