# The Smalltalk Report

The International Newsletter for Smalltalk Programmers

July-August 1993 Volume 2 Number 9

## SMALLTALK DEBUGGING TECHNIQUES

*By Roxie Rochat & Juanita Ewing*

Expert Smalltalk users are characterized not only by their programming skills, but by how quickly they locate and correct errors. Not only do they use debugging skills to find bugs, but also to understand existing code. To reuse code effectively, you have to understand it, so debugging skills are important tools for maximizing reuse and minimizing work.

This article describes debugging techniques for both Objectworks\Smalltalk and Smalltalk/V. Although written for novice Smalltalk users, it assumes a basic familiarity with Smalltalk terminology and the environment, including browsers and debuggers.

Many expressions in this paper are common to Objectworks and Digitalk Smalltalk systems. Expressions that are not annotated apply to both Smalltalk systems. Unless otherwise noted, the Objectworks expressions given in this article are applicable for:

- Objectworks\Smalltalk 4.1
- VisualWorks 1.0
- ENVY/Developer r1.41a for Objectworks\Smalltalk and VisualWorks.

The Smalltalk/V expressions have been tested under:

- Version 2.0 of Smalltalk/V for OS/2
- Version 2.0 of Smalltalk/V for Windows
- ENVY/Developer r1.41a for Smalltalk/V for Windows

If you are using other versions of Smalltalk, use the expressions presented in this article as a starting point.

### WHO AM I?

A major component of the debugging process is the collection of information about the objects and their current state. Transcript messages allow you to gather information about objects over time. Inspectors allow you to see objects and their internals in a static state. Careful planning with respect to naming and object identity can help you focus on easily collecting relevant information. This section reviews debugging techniques involving the Transcript, inspectors, and factors relating to object identity.

#### hello world, or printf, in Smalltalk

The Smalltalk equivalent of the C printf function is to write to the Transcript window. (You *do* leave your Transcript open, don't you? The system writes error messages to the Transcript so if you collapse or close it, sooner or later, you'll be sorry.)

Information you print in the Transcript can be used to determine when a particular method is called, to examine arguments, or to examine data calculated by

*continued on page 18...*

**Contents:**

*John Pugh*

*Paul White*

# EDITORS' CORNER

Developers who use Smalltalk have always had a real love/hate relationship with their development environment. We've always been fascinated listening to Smalltalkers describe the toolset in their environment. When describing Smalltalk to "outsiders," they defend it with an emotional fervor, noting how flexible it is and how rich a toolset it actually provides. But if you have a chance to speak with these same people alone, you'll hear a very different story. The fact is that the base Smalltalk development environment is in desperate need of a major overhaul. It has become Smalltalk's "legacy system." One of the first things that appealed to us about Smalltalk back in the early days was its rich development environment—it was definitely the best on the block. Since then, no significant changes have been made to the way in which people interact with the system. Sure, minor improvements have been made at times, but there have been no qualitative improvements to the browser, the inspector, or the debugger. Even the tools that do exist need to be more polished. (Ever listen to someone use the "Find Class" option in Digitalk's browser—the groans over a lack of wild card are universal). Even team development tools such as Team/V and ENVY don't improve to any significant extent the way in which we interact with Smalltalk.

So why don't we see better toolsets coming to market? We suspect the answer is simple: a lack of motivation on the part of the vendors. There is a greater return to be made by providing add-on facilities such as interface builders and database interfaces than there is by augmenting the tools that already exist in the base image. Will third-party developers take up the challenge? We hope so, but we are not terribly optimistic. Perhaps the forthcoming Object Explorer tool form First Class Software, which attempts to visualize the relationships between objects, will set a trend. Of course, many Smalltalk programming shops have built "in-house" extensions to the environment that they use on their projects and those of their clients. But most organizations don't want to be tool builders, they're application developers.

On a more positive note, four of the articles in this issue do illustrate just how rich an environment Smalltalk has. Each takes a different perspective, with two focusing directly on the debugging process and the techniques that can be used to understand what is taking place inside your systems. Roxie Rochat and Juanita Ewing are featured this month with their hints on debugging. They have included a number of debugging techniques, including ones for debugging code that does not allow for the normal "self halt" approach to work.

Also on the topic of debugging, Bob Hinkle, Vicki Jones, and Ralph Johnson return this month with a description of how Smalltalk can be extended in ways that will allow for non-intrusive debugging to be carried out.

Alan Knight and Kent Beck also touch on the issue of debugging with Smalltalk. Kent returns to his discussion of the conflicting roles played by inheritance in Smalltalk and introduces two new patterns that describe rules that can be applied when making inheritance decisions. Alan tackles the issue of recognizing "good code" by characterizing the elements of good coding techniques.

Finally, Richard Peskin provides us with a glimpse into work that is being done to make Smalltalk more applicable to scientific and engineering computing. As he points out, this area has not received much attention from the Smalltalk community lately, even though much of Smalltalk's early history involved serving this community.

John Pugh [illegible]

The Smalltalk Report (ISSN# 1056-7976) is published 9 times a year, every month except for the Mar/Apr, July/Aug, and Nov/Dec combined issues. Published by SIGS Publications Group, 588 Broadway, New York, NY 10012 212.274.0640. © Copyright 1993 by SIGS Publications, Inc. All rights reserved. Reproduction of this material by electronic transmission, Xerox or any other method will be treated as a willful violation of the US Copyright Law and is flatly prohibited. Material may be reproduced with express permission from the publishers. Mailed First Class. Subscription rates 1 year, (9 issues) domestic, $65, Foreign and Canada, $90, Single copy price, $8.00. POSTMASTER: Send address changes and subscription orders to: The Smalltalk Report, Subscriber Services, Dept. SML, P.O. Box 3000, Denville, NJ 07834. Submit articles to the Editors at 91 Second Avenue, Ottawa, Ontario K1S 2H4, Canada. For service on current subscriptions call 800.783.4903. Printed in the United States.

**The Smalltalk Report**

**Editors**
John Pugh and Paul White
Carleton University & The Object People

**SIGS Publications Advisory Board**
Tom Atwood, Object Technology International
Grady Booch, Rational
George Bosworth, Digitalk
Brad Cox, Information Age Consulting
Chuck Duff, Symantec
Adele Goldberg, ParcPlace Systems
Tom Love, Consultant
Bertrand Meyer, ISE
Meilir Page-Jones, Wayland Systems
Sesha Pratap, CenterLine Software
Bjarne Stroustrup, AT&T Bell Labs
Dave Thomas, Object Technology International

**The Smalltalk Report Editorial Board**
Jim Anderson, Digitalk
Adele Goldberg, ParcPlace Systems
Reed Phillips, Knowledge Systems Corp.
Mike Taylor, Digitalk
Dave Thomas, Object Technology International

**Columnists**
Kent Beck, First Class Software
Juanita Ewing, Digitalk
Greg Hendley, Knowledge Systems Corp.
Ed Klimas, Linea Engineering Inc.
Alan Knight, The Object People
Eric Smith, Knowledge Systems Corp.
Rebecca Wirfs-Brock, Digitalk

**SIGS Publications Group, Inc.**
Richard P. Friedman
Founder & Group Publisher

**Art/Production**
Kristina Joukhadar, Managing Editor
Susan Culligan, Pilgrim Road, Ltd., Creative Direction
Karen Tongish, Production Editor
Gwen Sanchirico, Production Coordinator
Robert Stewart, Computer Systems Coordinator

**Circulation**
Stephen W.Soule, Circulation Manager
Ken Mercado, Fulfillment Manager

**Marketing/Advertising**
James O. Spencer, Director of Business Development
Jason Weiskopf, Advertising Mgr—East Coast/Canada
Holly Meintzer, Advertising Mgr—West Coast/Europe
Helen Newling, Recruitment Sales Manager
Sarah Hamilton, Promotions Manager—Publications
Jan Fulmer, Promotions Manager—Conferences
Caren Polner, Promotions Graphic Artist

**Administration**
David Chatterpaul, Accounting Manager
James Amenuvor, Bookkeeper
Dylan Smith, Special Assistant to the Publisher
Claire Johnston, Conference Manager
Cindy Baird, Conference Technical Manager

Margherita R. Monck
General Manager

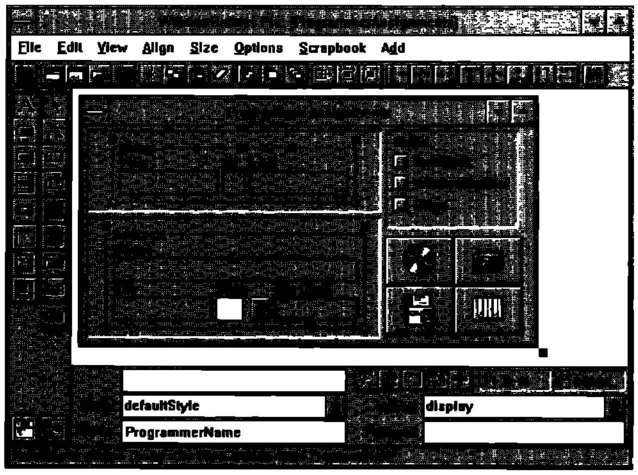
SIGS PUBLICATIONS

SHARE
OBJECT SYSTEMS
INC.

# WINDOWBUILDER PRO!

*The New Power in Smalltalk/V Interface Development*

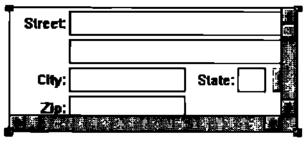
Smalltalk/V developers have come to rely on WindowBuilder as an essential tool for developing sophisticated user interfaces. Tedious hand coding of interfaces is replaced by interactive visual composition. Since its initial release, WindowBuilder has become the industry standard GUI development tool for the Smalltalk/V environment. Now Objectshare brings you a whole new level of capability with WindowBuilder Pro! New functionality and power abound in this next generation of WindowBuilder.



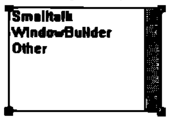
WindowBuilder Pro/V is available on Windows for $295 and OS/2 for $495. Our standard WindowBuilder/V is still available on Windows for $149.95 and OS/2 for $295. We offer full value trade-in for our WindowBuilder customers wanting to move up to Pro. These products are also available in ENVY®/*Developer* and Team/V™ compatible formats. As with all of our products, WindowBuilder Pro comes with a 30 day money back guarantee, full source code and no Run-Time fees.

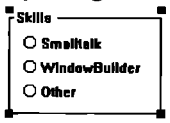
## *Some of the exciting new features...*

• CompositePanes: Create custom controls as composites of other controls, treated as a single object, allowing the developer higher leverage of reusable widgets. CompositePanes can be used repeatedly and because they are Class based, they can be easily subclassed; changes in a CompositePane are reflected anywhere they are used.

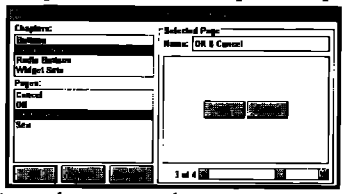


• Morphing: Allows the developer to quickly change from one type of control to another, allowing for powerful "what-if" style visual development. The flexibility allowed by morphing will greatly enhance productivity.

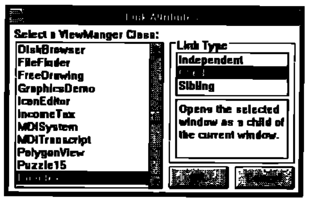


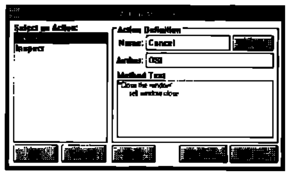


• ScrapBook: Another new feature to leverage visual component reuse, ScrapBooks provide a mechanism for developers to quickly store and retrieve predefined sets of components. The ScrapBook is a catalog of one's favorite interface components, organized into chapters and pages.



• Rapid Prototyping capabilities: With the new linking capabilities, a developer can rapidly prototype a functional interface without writing a single line of code. LinkButtons and LinkMenus provide a powerful mechanism for linking windows together and specifying flow of control. ActionButtons and ActionMenues provide a mechanism for developers to attach, create, and reuse actions without having to write code. These features greatly enhance productivity during prototyping.





• ToolBar: Developers can Create sophisticated toolbars

just like the ones in the WindowBuilder Pro tool itself.

• Other new features include: enhanced duplication and cut/paste functions, size and position indicators, enhanced framing specification, Parent-Child window relationship specification, enhanced EntryField with character and field level validation, and much more...

• Add-in Manager: Allows developers to easily integrate extensions into WindowBuilder Pro's open architecture.

Catch the excitement, go Pro!
Call Objectshare for more information.

**(408) 727-3742**

**Objectshare Systems, Inc**
**Fax: (408) 727-6324**
**CompuServe 76436,1063**

**5 Town & Country Village**
**Suite 735**
**San Jose, CA 95128-2026**

WindowBuilder and WindowBuilder Pro are trademarks of Objectshare Systems, Inc. All other brand and product names are registered trademarks of their respective companies

# Debugging objects

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

As the premier object-oriented programming language, Smalltalk should give programmers easy access to objects. However, during debugging it can 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 particular object, rather than all objects in a given class.This approach of defining only object-specific methods is similar to what Kent Beck has described.[1,2] 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 articles[3,4]) how powerful Smalltalk's reflective features are, as they allow programmers to adapt and extend the environment to suit their needs. The solution described here 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, although 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's method dictionary. To change a method for a 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-handed: Instances of Class have many instance variables and a lot of behavior 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 chapter titled "Protocol for Classes" in Reference 5.) 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's 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's 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's 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's 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 lookup 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.

* 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.

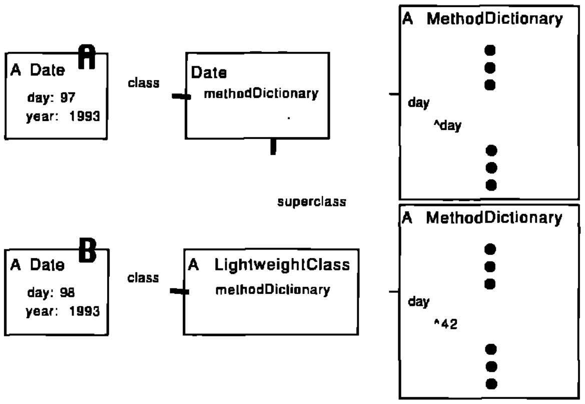


Figure 1. The relationship between an object, its original class, and its lightweight class.

In 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's 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 that 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.) 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 classquote 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

EC-Charts

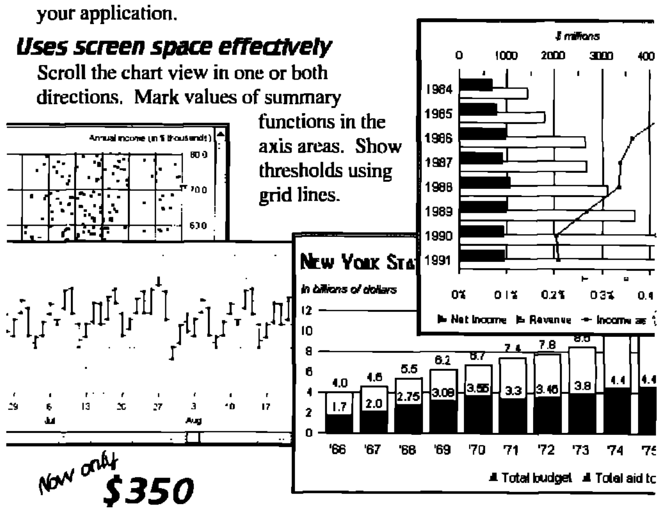
***Just touch a button to put a chart view in your window!***

***Add charts to your VisualWorks palette***

***Dynamic*** Add or change data points, with minimal screen repainting. Add or remove data series to/from the chart.

***Interactive*** Select data points with the mouse—EC-Charts informs your application.

***Uses screen space effectively*** Scroll the chart view in one or both directions. Mark values of summary functions in the axis areas. Show thresholds using grid lines.



Now only **$350**

No runtime license fee

***Call for a technical paper on EC-Charts***

East Cliff Software

**(408) 462-0641**

21137 East Cliff Dr · Santa Cruz · CA 95062

VisualWorks is a trademark of ParcPlace Systems, Inc.

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 Reference 6.) 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 as the lightweight class's 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 overly 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.

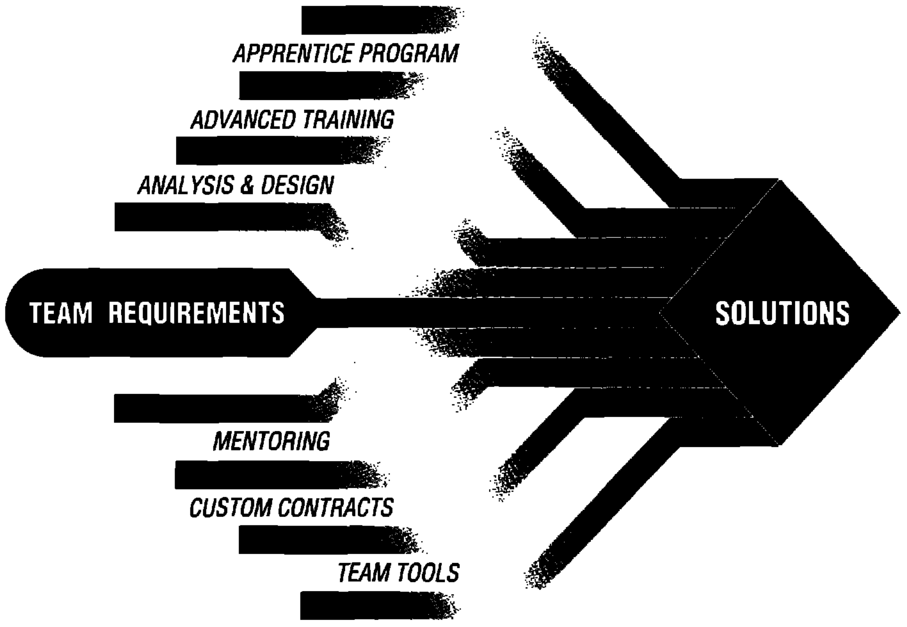
## BREAKPOINTS

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 are 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 are invisible in source code, so a programmer who is 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 halt only 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 Break-

*continued on page 24...*

# Object Transition by Design



**Object Technology Potential**

Object Technology can provide a company with significant benefits:

- Quality Software
- Rapid Development
- Reusable Code
- Model Business Rules

But the transition is a process that must be designed for success.

**Transition Solution**

Since 1985, Knowledge Systems Corporation (KSC) has helped hundreds of companies such as AMS, First Union, Hewlett-Packard, IBM, Northern Telecom, Southern California Edison and Texas Instruments to successfully transition to Object Technology.

**KSC Transition Services**

KSC offers a complete training curriculum and expert consulting services. Our multi-step program is designed to allow a client to ultimately attain self-sufficiency and produce deliverable solutions. KSC accelerates group learning and development. The learning curve is measured in weeks rather than months. The process includes:

- Introductory to Advanced Programming in Smalltalk
- STAP™ (Smalltalk Apprentice Program) Project Focus at KSC
- OO Analysis and Design
- Mentoring: Process Support

**KSC Development Environment**

KSC provides an integrated application development environment consisting of "Best of Breed" third party tools and KSC value-added software. Together KSC tools and services empower development teams to build object-oriented applications for a client-server environment.

**Design your Transition**

Begin *your* successful "Object Transition by Design". For more information on KSC's products and services, call us at 919-481-4000 today. Ask for a FREE copy of KSC's informative management report: *Software Assets by Design.*

**Knowledge Systems Corporation**

OBJECT TRANSITION BY DESIGN

114 MacKenan Dr.
Cary, NC 27511
(919) 481-4000

# Applications of Smalltalk in scientific and engineering computation

*Richard L. Peskin*

1992 marked Smalltalk's 20th anniversary. While using Smalltalk for simulation was an important goal for the environment, applications to "real" scientific and engineering simulation and modeling have been few. In earlier Smalltalk systems, slow (and expensive) hardware together with slow interpreters were adequate reasons for the scientific community to ignore Smalltalk. Addiction to FORTRAN and conservatism compounded the problem.

Today's modern Smalltalk systems running on high performance workstations have removed some of the traditional barriers to the use of the language for scientific computing. While interpretive environments are generally an order of magnitude slower than optimized compiled code for numerically intensive tasks, techniques to integrate compiled code segments into Smalltalk applications can overcome this deficit. The advantages of Smalltalk's graphical interface and its ability to promote prototyping offer much for scientific computing.

To address the issues and problems presented by scientific applications of Smalltalk, Kent Beck of First Class Software and I organized a workshop at OOPSLA '92 in Vancouver. Attendance was by invitation only. Ten position papers were presented during the morning session, the afternoon session was devoted to informal workgroups that delved into design and implementation specifics. The position papers covered a wide range of domain-specific topics concerned with applying Smalltalk to scientific and engineering computation. However, all the papers were characterized by certain commonalities, one of these being that Smalltalk's flexibility does admit strategies to overcome weaknesses such as computational performance. I opened the meeting with some overview comments and noted the rising interest in object-oriented computing within the scientific and engineering community. Furthermore, with the rapid increase in hardware performance, we can expect more applications of interpretive environments to scientific and engineering problems. This is already evident in journal articles where languages like Lisp, Prolog, Smalltalk, etc. are taking place alongside FORTRAN and C. However, this domain community is very demanding; if existing O-O environments are not suitable, users will create ones that are. Sather is an example.

I also emphasized the need for robustness, completeness, and correctness in Smalltalk implementations if they are to meet the needs of the scientific community. Support for external programs, inter-application communication, distributed and parallel computation, and numerical and symbolic computation classes are just some of the features needed, but are currently either absent or minimally present in Smalltalk systems. This level of support may be a tall order for a language with only one or two vendors and no "standard"; one reason for the popularity of Lisp among the scientific community is its standards and its multi-vendor support.

The bottom line is that the scientific and engineering computation community will adopt O-O systems and do want the prototyping flexibility offered by an interpretive environment with dynamic binding. If Smalltalk is to be chosen by more than just a token few, its user community and vendors will have to work together to meet the needs of scientists and engineers. The OOPSLA workshop was set up to be one forum to assist in this process. To this end, vendor representatives were invited to attend, and ParcPlace Systems had a representative at the workshop. The morning presentations were further divided into general topics (mathematics, engineering computation, scientific computation, and scientific data management) and application papers. However, these boundaries were not sharp. Professor David Rector of the University of California, Irvine opened the morning session with a discussion of his work in the development of a Smalltalk-based system to teach numerical analysis to students. He presented several examples of how current Smalltalk standard implementations fail to provide needed support. One example is the absence of precise interval subdivision (which he has corrected). He suggested implementing a new iterator, map: [aBlock], so that collection operations return correctly (e.g., so that collect: over a dictionary returns a dictionary, etc.), and he showed how this applies to a differential equation solver method. Rector suggested a separate class, Quantity, under Object, because Number is not appropriate to hold integral domains and fields such as complex numbers, polynomials, quaternions, etc. He also pointed out that class Array is not the proper container for Vectors and Matrices. In particular, the many varieties of matrices implies the need for a more general class to deal with these objects. This subject became the topic of one of the afternoon working groups.

Alan Knight, formerly of the Department of Mechanical and Aerospace Engineering at Carleton University, presented an overview (co-authored with N. Dai) of Smalltalk in the contexts of applications to finite element method solvers. Drawing on five years of experience in attempting to use Smalltalk for this type of problem, he listed the major problem areas of performance, portability, graphics, and user-interface facilities.

Approaches to performance improvement include use of primitives and high-performance libraries, and improved implementations. Knight pointed out that Smalltalk's claimed high portability falls short of the mark in practice, both in portability between versions and limited number of supported platforms. Smalltalk's integration with other languages needs to be improved, as do graphics (particularly 3-D graphics) for scientific and engineering applications. The integration and graphics issues were also discussed in other papers at the workshop. Weaknesses in the user interface, particularly the need for good widget toolkits was mentioned, and he emphasized the need for significant improvements in the debugger.

Dr. Rob Gayvert of RIT Research Corp. discussed the use of Smalltalk in scientific computations, with emphasis on applications in speech and signal processing. He also emphasized the need for improvement in the numeric array and matrix classes, listing specific new protocols for both numeric array and matrix classes. His group has implemented these in Smalltalk/V Mac. His suggested strategy for domain-specific classes (such as may arise in nonlinear equation solvers) is to implement first without regard to performance and then to optimize. The RIT group has implemented inter-application communication (specifically AppleEvents) as well as extensions to the ToolBox access in Smalltalk/V Mac. This greatly increases the potential for access to external data sources, application servers, etc. This should be a standard feature in future Smalltalk releases. Gayvert showed examples of his system improvements, namely the speech processing application. Better numeric and matrix classes, IAC, etc. allowed the construction of tools to do speech processing, which have both algorithmic power and good graphical presentation for the user. His conclusion is that, with proper additions and improvements, Smalltalk has strong potential for scientific and engineering applications.

Dr. Sandra Walther of Rutgers, in a paper I co-authored, reviewed some features of the Smalltalk-based SCENE system, a software environment to support numerical experimentation in science and engineering. Some features of importance in this environment include user extensibility and configurability, automatic programming, computational steering, distributed storage, and parallel/distributed processing. The talk focussed on the strategies used to handle very large data sets—sets so large that their representation in Smalltalk as data objects is impractical. The large data sets were implemented as active processes running on a (server) platform. In this way, one can handle these sets efficiently, but to users of the Smalltalk interface the sets appear as manipulatable objects. Practical use of this scheme requires some good interprocess communications, and a means for users to tailor particular data sets to meet their needs. The latter facility is provided by an *object editor* tool that is used to create and compile new C code for the active data set and tailor menus and other interface items in response to user directives.

In conclusion, Smalltalk can be appended to handle large data sets and other scientific computational requirements.

**ODBMS**

**ODBMS**
**The Objectoriented Database**

- ☐ Persistent Object Storage for Smalltalk
- ☐ Handles Complex Data Types
- ☐ Object Ownership, Versioning, Security, and Object Distribution
- ☐ Programmer and Enduser Versions
- ☐ Stand Alone or Network Configuration
- ☐ Database Classes licensed for OEM Distribution
- ☐ Support for ParcPlace Smalltalk-80

**Add-on Applications**
- ☐ DSSDe SourceCode Management
- ☐ Interface to SQL-Classes
- ☐ Support for Digitalk's PARTS

**ODBMS**
**Objectoriented Technology by**
**VC Software**

**USA: VC Software Inc., Three Christina Centre, 201 N.Walnut Street, Suite 1000, Wilmington, DE 19801 <> Other Countries: VC Software Construction GmbH, Petritorwall 28, 38118 Braunschweig, Germany, Tel: +49-531-24 24 00, Fax: +49-531-24 24 0-24**

These facilities provide Smalltalk-like incremental compilation and dynamic binding features outside of the actual Smalltalk environment.

The portion of the workshop devoted to applications began with a talk by Jan Steinman of Bytesmiths. He described his work in using Smalltalk to develop laboratory instrumentation interfaces. He introduced the concept of the "abstract" instrument object (instances of an InstrumentObject class), which allow standard abstractions of physical instruments and effects a basis for common data acquisition protocols. Other features, such as appropriate abstract protocols, were also discussed. As an example he described the Tektronix instrument ensemble control system, a stack-based machine architecture for controlling instruments and returning results via a graphical interface. This was developed under the object paradigm in Smalltalk. The position paper by P. Johnson and D. Herkimer of Martin Marietta was not presented, but copies were available. The paper describes a space vehicle launch simulator written in Smalltalk/V Mac. Among the issues discussed were the need for support for parallel computation abstractions in Smalltalk that would provide a framework for implementation of parallel computation of numerically intensive portions of these complex simulations. This paper also pointed out the need for better numerical classes in Smalltalk. Brian Remdeios of BC Research presented a Smalltalk application designed to simulate control functions for an IC engine. The hierarchical nature of class structure allows encapsulation of various engine component parts into a single functional representation or the ability to study individual components. In this application, Smalltalk was able to facilitate inter-object communication, but it was suggested that a class to handle more general transfer functions between objects would be helpful. The paper discussed how Smalltalk models of this type could be used to implement non-brittle (e.g., fuzzy logic) decision process simulations.

David Jones of Prior Data Science presented a paper on algorithm objects. While the specific application discussed was taken from the domain of geometric models, this paper presented a controversial proposal, namely, to collect algorithms (methods) under a single class (Class Algorithm). This is a radical departure from current Smalltalk practice where algorithmic methods are associated with specific class behaviors. Under the Class Algorithm proposal, algorithms together with the their documentation etc. would be found in a single class, supported by its own browser and other interface features. Users would have a single point of reference for all algorithms, and class behaviors would be implemented via dispatch from Class Algorithm. This proposal was the subject of one of the afternoon working groups.

Judith Cushing of the Oregon Graduate Institute discussed the subject of computational proxies. The difficult issue here is how to render results computed by different scientific programs comparable. The emphasis in this paper was on the computational chemistry domain, but the central issue of how to design object-oriented databases that can capture both syntactic and architectural complexity associated with the output of various scientific computational systems all of which produce data relevant for a given domain experiment or simulation. Implementation approaches in C++ were discussed, and these were related to possible Smalltalk implementations.

**"Modern Smalltalk systems running on high performance workstations have removed some of the traditional barriers to the use of the language for scientific computing."**

The final paper in the first session of the workshop was presented by Annick Fron of DEC European Technical Center in France. She described an interesting application of Smalltalk to the simulation of an MIMD embedded computer system. The simulation relied on processes and monitors. The result is a tool that has been used for embedded signal processing applications. This type of tool is very useful in design and debug stages and can ease problems associated with integration on final target architectures.

The afternoon sessions were devoted to in-depth considerations of topics that arose during the presentations. Informal groups examined issues such as the need for better mathematical algorithms and better organizations for algorithms, interfacing Smalltalk to parallel and distributed computing, and mechanisms for handling scientific data in Smalltalk environments. Suggestions from these sessions included the need to re-examine algorithms and algorithm classes, the need for better integration of Smalltalk into scientific computing environments, the need for better class support for parallel and distributed computing interfaces, etc. One important conclusion of the workshop was that this event should be repeated, perhaps on a regular basis. There was a general feeling that the scientific and engineering community was ready for Smalltalk. The critical question is whether Smalltalk is ready for that community. ■

*Richard L. Peskin is Professor of Mechanical and Aerospace Engineering at Rutgers University where he is director of the CAIP Center Computational Engineering Systems Lab. He has been involved with engineering and scientific aspects of Smalltalk since 1984. He is one of the designers of the SCENE (Scientific Computation Environment for Numerical Experimentation) system, a Smalltalk-based distributed computing environment that implements computational steering tools such as interactive scientific graphics and data management, automatic equation solvers, and mathematical expert systems. He can be reached via email at peskin@caip.rutgers.edu.*

Alan Knight

# Good code, bad hacks

There have been many attempts to define the elements of Smalltalk style. Some of them even agree with each other. Almost all of them share a common point of view, that of a programmer striving to write good code. Honna Segel (honna@bnr.ca), on the other hand, approaches the problem as someone evaluating a Smalltalk program, trying to recognize bad code:

> **I'm in the curious position of evaluating a prototype written in Smalltalk without prior knowledge of Smalltalk. I could distinguish a terrible hack from good work in C—what do I look for in Smalltalk? What's a prime symptom of work that will be scary to modify and extend?**

## THE BASICS

Dan Benson (benson@siemens.siemens.com) writes:

> **As a first pass, I'd look at the class hierarchy. See if the names of the classes match the concepts intended for the prototype. For instance, if the prototype is supposed to be an airline reservation system you might expect to find classes representing Tickets, Airlines, Reservations, Airports, and so on. If the class names are way off the mark, I would be a bit skeptical. Next, see if there are any class comments to see whether the programmer was conscientious or at least considered that someone else might read the code.**
>
> **Some of the other things you can look for without getting into actual code are the organization of the class hierarchy (to see if it makes sense intuitively), the method categories (to see how well the various tasks were separated), and, perhaps, the number of instance variables and the names used (there shouldn't be too many instance variables per class, and the names should be intuitive or at least informative).**
>
> **The most obvious thing to check, of course, is the operation of the prototype itself. How well does it do what it's supposed to do? Are there any bugs? If so, how serious are they? Is it a matter of changing the interface or would it involve modifying the underlying model, or perhaps starting all over?**

There's good advice here, and most of it can be applied by someone who doesn't know Smalltalk well. Coincidentally, I've actually seen an airline reservation system written in Smalltalk that did not have classes representing Tickets, Airlines, Reservations, or any of the other obvious domain objects. Sure enough, it was bad code.

One of these remarks, however, does seem questionable to me. We are to check to see if the class hierarchy "makes sense intuitively." That's pretty vague, especially for someone who's unfamiliar with Smalltalk. While the hierarchy *should* make sense intuitively, this suggestion needs to be defined more clearly.

For myself, I would say that classes in an inheritance hierarchy should have a clear logical relation. This relation should probably be expressible as either "is-a" or "is-implemented-like." This is not a two-way relationship. Not all classes that have these relationships should be in the same inheritance tree.

This still leaves much room for judgment, as it should, but I hope it helps weed out some of the worst offenders (such as those using the "sounds-like" or "was-implemented-the-same-day-as" relations to determine their class hierarchies).

## DOCUMENTATION

Jack Woehr (jax@well.sf.ca.us) has a simple recipe:

> **Good Smalltalk comes accompanied by good documentation, a separate document explaining the author's intent, and probably by a glossary of objects and their methods.**
>
> **Bad Smalltalk comes without such documentation.**

Strictly speaking, the quality of the documentation and the quality of the code should be independent. If you take away the documentation, the quality of the code remains the same. All of us have written good code that we never quite got around to documenting properly.

Practically speaking, however, good code and good documentation are inseparable. This is especially true for code that tries to be reusable (and these days, we're all writing reusable code). When I intend to use a class, the first thing I do is look for the class comment. All too often, the second thing I do is curse the author for not providing one.

ParcPlace, to its credit, provides comments for all of its system classes. Digitalk doesn't support class comments directly, but it's easy to establish a convention for class methods containing comments.

## OTHER CRITERIA

Frerk Meyer (frerk@tk.telematik.informatik.uni-karlsruhe.de) provides a whole list of criteria. His suggestions are somewhat more difficult for novices to apply and subject to some exceptions. I'll discuss them one at a time.

### Use Global Variables Sparingly

Bad—the use of global variables

This is pretty standard, even for non–O-O programming. Globals have their uses, but they definitely should not be used to excess because they introduce extra dependencies between classes and generally pollute the namespace.

### Separate Domain and Interface

Bad—instance variables in the model holding view, controller, or window information

This is ParcPlace-specific, but the underlying idea is universal. The domain model should not concern itself with the way in which the interface presents information. While this is very important, it is something that may be difficult for Smalltalk novices to judge and difficult for Smalltalk programmers to do well.

The simplest method of checking for this separation is to examine the instance variables and methods of the domain model for obvious interface information. This will find some violations, but assumptions about the interface can leak into the domain model in many subtle ways. There's always a temptation to introduce just a few lines of code that are ever-so-slightly dependent on the interface. Maybe it doesn't really belong in the interface, either. Besides, it would take so much longer to do it properly, and we're not likely to change that part of the interface.... These temptations should be resisted.

Greg Hendley and Eric Smith discussed these issues in some detail in a two-part article titled "Separating the GUI from the application" (THE SMALLTALK REPORT, May 1992 and October 1992). They advocate introducing a "control" layer into the interface that acts as a buffer between the interface visuals and the domain model.

### Avoid Long Methods

Bad—methods that are larger than one screen (usually)

It's pretty much a consensus that Smalltalk methods should be short. Long methods are probably trying to do more than one thing and should be broken up into their components. Long methods aren't always bad, but the presence of large numbers is a definite danger sign.

A notable exception is for automatically generated methods, such as WindowBuilder's horrendously long open methods. But since these methods are not intended to be modified by humans, this is not so much of a problem.

I notice that Digitalk's compiler is much slower for long methods. This can, however, be considered a *feature* (though I doubt it was intended as one) since it motivates programmers to break up their code into smaller components.

### Avoid System Changes

Bad—making changes to system classes instead of subclassing

After some discussion, the consensus on this point was that adding methods to system classes is fine, but modifying existing methods is to be avoided. System changes are a problem because your changes are likely to be incompatible with others, including those in the next Smalltalk version. They're also more likely to make your system crash during development. If you have to modify a system method, it's usually best to make the modification as small as possible. Ideally, you should just insert a hook that calls your own code.

### Keep Instance Creation Simple

Bad—using class method **new** more than **^super new initialize**

It's common practice in Smalltalk to override the method new to automatically initialize instances of the class, changing the code to:

```
new
    ^super new initialize
```

Other common changes are to override new to be an error or to return an already existing instance. Adding much more functionality than this to the method is considered bad form. Again, it's better to provide a hook to more extensive code in a method like initialize.

### Use System Classes

Good—using system classes wherever possible

If code that serves a purpose is already available, it should be reused. As an extreme example, code that uses fixed-size arrays, but goes through complex manipulations to mimic the behavior of OrderedCollection would be bad. Similarly, code that avoids the normal user interface mechanisms and gets mouse or keyboard input directly is probably bad. It may be trying to do something that is not normally possible through those mechanisms, but even then it is preferable to extend the UI mechanisms rather than go around them.

### Work within the System

Good—using MVC, dependency mechanisms, and processes

Again, if the mechanisms are there, it's best to work with them rather than against them. They can, however, be overused. Kent Beck writes, in "Abstract Control Idioms" (THE SMALLTALK REPORT, July/August 1992), about the advantages and disadvantages of the dependency mechanism.

He summarizes the disadvantages as "debugging and performance." Dependency-based code can be much more difficult to follow and debug than normal code. When it's put together properly, it will often work immediately. When it doesn't, tracking down the problem can be painful.

I wouldn't consider processes to be a necessary feature of good code. Multi-threaded code introduces many complications, and I avoid it unless I really need it.

### Choose Names Carefully

**Good—using expressive naming of classes, methods and variables, and using the class document feature**

Definitely. Naming things properly is very important. One of my biggest complaints about both Digitalk and ENVY/Developer is how difficult they make it to change class names.

## PUT CODE IN THE RIGHT PLACE

Charles Lloyd (clloyd@gleap.jpunix.com) adds several points.

### Place Code Well

**A series of messages sent to some object other than self is probably badly placed code. That series should be moved to the class of the receiver.**

**Note: This is the hardest thing to do well in O-O programming, but it pays very high dividends when done well.**

Breaking up methods in this way has several advantages. As we've already mentioned, it's good practice to break up long methods into logically connected units. A series of messages to some other object makes a good candidate for such a division. Since they have an object in common, they should probably be moved to a method in its class. This also provides an opportunity to use polymorphism (i.e., providing different implementations of the same function in other classes).

### Avoid Checking Types Explicitly

**Encoding type information**

**You should never see any checks for "type" information. All type information should be implicit in the class of the receiver. Exceptions to this rule are few and far between.**

It's usually bad style to ask the type of an object. Frequent use of class tests or isKindOf: is a characteristic of poor code.

Ideally, rather than testing the type, code should request that an object carry out some action. The object is then responsible for doing the appropriate thing based on its type, but this is done through the method dispatch mechanism, rather than explicitly in code.

If it's necessary to determine some characteristic of the object, it's better to do so by sending a message asking about the characteristic. Thus, it's better to say:

```
anObject isCollection ifTrue: [ ... ]
```

than

```
(anObject isKindOf: Collection) ifTrue: [ ... ]
```

The second form confuses an attribute of the object (whether it responds to basic collection protocol) with the class hierarchy (whether it inherits from the class Collection).

As a concrete example of how this can be dangerous, consider a system that works with vectors. We may wish to treat instances of Point as two-dimensional vectors. Code that sends the message isVector will work fine for points. Code that relies on isKindOf: Vector will fail.

### Put Conditional Behavior in Subclasses

**Introduction of instance variables**

**Instance variables should be added sparingly. If you think you need *N* instance variables to model your subclass, consider introducing *M* subclasses (*M* very close to *N*) where each new subclass introduces a minimum of new variables.**

Introducing subclasses where other languages might use enumerated "type" variables is often good style. It is a problem if instances may change their type, but, otherwise, it can be very useful. In many ways, it's similar to the previous point. Instead of having conditional statements on the enumeration, we simply ask instances to perform some function. They will automatically do it in the appropriate way, and the language mechanisms will do the testing for free.

## FAILURE MODES

We can also look at bad code by considering how it might have gotten to be bad. Maybe the author didn't understand Smalltalk or OOP fully. Maybe it was a quick hack by someone capable of doing better work. Maybe it was written by someone who didn't understand the domain and/or requirements. Maybe it really was written by an idiot. Maybe it was once good code that's had too many patches and has never been consolidated.

Most of these problems can be recognized the same way they would be in any programming language, and only a few have OOP- or Smalltalk-specific aspects.

Quick hacks, for example, can usually be identified by their shoddy documentation and comments. The comments that do exist are often incomprehensible notes from authors to themselves, often of the form "fix this later."

It's usually easy to tell when the author didn't understand the paradigm and wrote FORTRAN, C, or COBOL with Smalltalk syntax. There is often excessive use of type information (as described above), internal representations are almost always exported, and collections with encoded meanings are often used as data structures.

The most common symptom of exporting too much representation is the presence of direct get/set methods for every variable in a class. Some schools of thought hold that all variable references should be made through get/set methods. In this case, the code will have such methods, but many of them should be clearly marked as private.

Programmers who aren't used to opaque data types will often use collections as data structures. For example, they might represent a circle by an array whose first element is the centre point and whose second is the radius, instead of introducing a new class Circle. Juanita Ewing discusses this common error in "Don't use Arrays?" (THE SMALLTALK REPORT, May 1993). ■

## CONCLUSION

Although it's far from complete, I hope this brief overview provides some help to those of you trying to distinguish good Smalltalk from bad Smalltalk. If you're writing code, this column should provide some things to strive for or avoid.

*Alan Knight works for The Object People, 509-885 Meadowlands Dr., Ottawa, Ontario, K2C 3N2. He can be reached at 613.225.8812 or as knight@mrco.carleton.ca.*

**ArtBASE®**

**Distributed Smalltalk** and **ODBMS**

**for VisualWorks™ and Objectworks®**

ArtBASE will never pretend that it is **the best ODBMS ...**

... but it is **the only distributed Smalltalk** environment still fully implemented in Smalltalk

- any object and class extended by the ability to become persistent and to be shared by multiple users
- full transaction management
- all advantages of Smalltalk kept alive
- almost no changes to existing applications to convert them to a database
- delivered in source code

ArtInAppleS

ArtInAppleS Ltd.
Kremelska 13
845 03 Bratislava
Slovakia

fax: +42-7-777 779
tel: +42-7-362 889
artbase@artinapples.cs

... just **add** ArtBASE source code to your Smalltalk application and you can **taste** the **flavor** of a **really distributed** Smalltalk environment

free evaluation licenses available

Kent Beck

# Inheritance: the rest of the story

Of the three tenets of objects—encapsulation, polymorphism, and inheritance—inheritance generates by far the most controversy. Is it for categorizing analysis objects? Is it for defining common protocols (sets of messages)? Is it for sharing implementation? Is it really the computed goto of the nineties?

The answer is Yes. Inheritance can (and does) do all of the above at different times. The problem comes when you have a single-inheritance system like Smalltalk. You get one opportunity to use inheritance. If you use it in a way that doesn't help you, you have wasted one of the most powerful facilities of the language. On the other hand, if you use it poorly, you can mix up the most ridiculous, unmaintainable program gumbo you've ever seen. How can you walk between the rocks of under-using inheritance and the chasm of using it wrongly?

What's the big deal? Inheritance is the least important of the three facilities that make up objects. You can do valuable, interesting object-oriented programming without using inheritance at all. Programmers still quest after the Holy Grail of inheritance because of the potential it shows when it works well. When you need an object and find one that is factored well and does almost what you want, there are few experiences in programming better than making a subclass and having a working system after writing two or three methods.

In this and my next several columns, I will focus on various aspects of inheritance. I will present a variety of strategies for taking advantage of inheritance, in the form of patterns. While I don't necessarily use all the patterns in my own programming, casting the strategies in terms of patterns makes it easier to compare and contrast them.

## PATTERN: COMPOSE METHODS

This pattern is the cornerstone of writing objects that can be reused through inheritance. It is also critical for writing objects that you can successfully performance tune. Finally, by forcing you to reveal your intentions through method names, it makes your programs more readable and maintainable.

### Context

You have some code that behaves correctly (it does no good to beautify code that doesn't work, unless you have to make it work). You go to subclass it, and realize that to override a method you have to textually copy it into the subclass and change a few lines, forcing you forever after to change both methods.

Another good context for this pattern is when you are looking at a profile that looks flat; that is, no single method stands out as taking more time than others. You need further improvement in performance and believe that the object can deliver it.

### Problem

How can you write methods that are easy to override, easy to profile, and easy to understand?

### Constraints

Fewer, larger methods make control flow easy to follow. Lots of little methods make it hard to understand where any work is getting done. Lots of little methods named by what they are intended to do, not how they do it, make understanding the high-level structure of a computation easy. Your programming time is limited. You only want to perform manipulations of the code that will have some payoff down the road. Each message sent costs time, and execution time is limited. You only want to cost yourself execution time if the result will provide some advantage at some point. You don't want to introduce defects in working code. The manipulations must be simple and mechanical to avoid errors as much as possible.

### Solution

Make each method do one nameable thing. If a method does several things, separate out one of them, create a method for it, and invoke it in the original method. When you do this, make sure that if the same few lines occur in other methods, those methods are modified to invoke the new one as well.

This solution ignores the cost of message sending. You will get faster programs by using messages to structure your code so that you can more easily tune them than by reducing the number of messages. It also assumes that the eventual reader of the code is comfortable piecing together control as it flows through lots of small methods.

### Example

A method for parsing a stream to eliminate lines that begin with a pound sign might look like this at first:

```
parse: aStream
    | writer |
    writer := String new writeStream.
    [aStream atEnd] whileFalse:
        [(aStream peekFor: $#)
            ifTrue: [aStream restOfLine]
            ifFalse: [writer nextPutAll: aStream restOfLine]]
```

Applying "Compose Methods" to parse: to separate line parsing from the overall parsing control structure we get:

```
parse: aStream
    | writer |
    writer := String new writeStream.
    [aStream atEnd] whileFalse:
        [self parseLine: aStream onto: writer]

parseLine: inStream onto: outStream
    (aStream peekFor: $#)
        ifTrue: [^aStream restOfLine].
    outStream nextPutAll: aStream restOfLine
```

Notice that by creating parseLine:onto: we are now able to use the return control structure to make the submethod easier to extend. Applying it again to factor out the outputStream creation, we get:

```
parse: aStream
    | writer |
    writer := self outputStream.
    [aStream atEnd] whileFalse:
        [self parseLine: aStream onto: writer]

outputStream
    ^String new writeStream
```

Applying it to parseLine:onto: to separate the choice of what is a comment from the behavior when a comment is found we get:

```
parseLine: inStream onto: outStream
    (self peekForComment: inStream)
        ifTrue: [inStream restOfLine].
    outStream nextPutAll: inStream restOfLine

peekForComment: aStream
    ^aStream peekFor: $#
```

Apply it to peekForComment: to separate the character you are looking for from the way in which you look for it:

```
peekForComment: aStream
    ^aStream peekFor: self commentCharacter

commentCharacter
    ^$#
```

The final code is much easier to modify in a subclass if you want to change the comment character, write onto something other than a string, or extend the parsing to deal with special cases other than comments.

## PATTERN: SEPARATE ABSTRACT FROM CONCRETE

This is a pattern I learned from Ken Auer of Knowledge Systems Corporation. He told me about using it to great advantage in a financial services application in which there were many kinds of financial instruments, all implemented similarly.

“By understanding the options and trade-offs involved, you can use it to your advantage.”

### Context

You have implemented one object. It has some methods that rely on the values of variables, and others that do not. You can see that you will have to implement many other similar objects in the future.

### Problem

How can you create an abstract class that will correctly capture the invariant part of the implementation of a family of objects with only one concrete example?

### Constraints

You want to begin using inheritance as early as possible to speed subsequent development, and you want you inheritance choices to be correct so you don't have to spend time refactoring later.

### Solution

Create a state-less superclass. Make it the superclass of the class you want to generalize. Put all of the methods in the subclass which don't use variables (directly or through accessors) into the superclass. Leave methods that rely on instance state in the subclass.

This solution strikes a balance between inheriting too early and too late. By making sure you have one working class you know you aren't using inheritance entirely on speculation.

### Example

Let's say that we have an RGBColor represented as red, green, and blue values between 0 and 1. We can then write methods like:

```
hue
    "Complicated code involving the instance variables red, green, and
    blue..."
saturation
    "Complicated code involving the instance variables red, green, and
    blue..."
value
    "Complicated code involving the instance variables red, green, and
    blue..."
complement
    ^self species
        hue: (self hue + 0.5) fractionalPart
        saturation: self saturation
        value: self value
```

Applying "Separate Abstract from Concrete" to RGBColor, we create Color as RGBColor's superclass. We move complement to Color, because it doesn't rely on any instance variables directly. We leave hue, saturation, and value in RGBColor because they do rely on variables.

Now, if we want to create Color subclasses that store color values in other ways, they can inherit complement as long as they implement hue, saturation, and value.

When you apply this pattern, you will often find that methods which were implemented initially as requiring variable values can be recast by applying "Compose Methods" so they can be moved into the superclass.

## CONCLUSION

Now that I have written down *Separate Abstract from Concrete*, I'm not sure I entirely agree with it. I like to have more than one concrete example before I try to generalize. I use two different patterns, "Factor Several Classes" and "Concrete Superclass" in my own programming. I will present these patterns in the next issue.

Inheritance is strong medicine. Only by understanding the options and trade-offs involved can you avoid the pitfalls and use it to your advantage. If you use different patterns for applying inheritance, please send them to me. ■

*Kent Beck is founder of First Class Software. He can be reached at 408.338.4649 (v), 408.338.3666 (f), or via CompuServe at 70761,1216.*

VOSS

**Virtual Object Storage System for**

***Smalltalk/V***

*Seamless persistent object management for all Smalltalk/V applications*

- Transparent access to all kinds of Smalltalk objects on disk.
- Transaction commit/rollback of changes to virtual objects.
- Access to individual elements of virtual collections for ODBMS up to 4 billion objects per virtual space; objects cached for speed.
- Multi-key and multi-value virtual dictionaries for query-building by key range selection and set intersection. Partial and concatenated keys supported.
- Works directly with third party user interface & SQL classes etc.
- Class Restructure Editor for renaming classes and adding or removing instance variables allows applications to evolve.
- Shared access to named virtual object spaces on disk; object portability between images. Virtual objects are fully functional.
- Source code supplied.

VOSS/OS2 $1950, VOSS/Windows $1950, VOSS/286 $950.
VOSS/OS2 DLL (excluding source code) $595.
'The VOSS Collection' - source code for non-virtual collections only, (Windows and OS/2 versions), with VOSS/OS2 Demonstration - $150.
Quantity discounts from 30% for two or more copies. (Ask for details)
Visa, MasterCard and EuroCard accepted. Please add $15 for shipping.

logic ARTS

Logic Arts Ltd 75 Hemingford Road, Cambridge, CB1 3BY England
TEL: +44 223 212392 FAX: +44 223 245171 CIS: 100040,364

## ■CALENDAR■

**July 16-19, 1993**
**OBJECT EXPO EUROPE**
*London, England*
*44.0.306.631.331*
*44.0.306.631.696 (fax)*

**July 19-23, 1993**
**IBM CONFERENCE ON OBJECT-ORIENTED SD TOOLS**
*Toronto, Canada*
*512.838.8019*

**August 2-10, 1993**
**DESTINATION C++**
*New York, NY*
*Washington, D.C.*
*Toronto, Canada*
*Chicago, IL*
*Houston, TX*
*Los Angeles, CA*
*212.274.9135*

**August 10-12,1993**
**SUN OPEN SYSTEMS WEST**
*Anaheim, CA*
*512.250.9756*

**Sept. 26-Oct. 1,1993**
**OOPSLA**
*Washington, D.C.*
*212.869.7440*

**September 21-23,1993**
**UNIX EXPO**
*New York, New York*
*800-829-3976*
*201-346-1602 (fax)*

**October 13-15,1993**
**INT'L SYMPOSIUM & EXHIBITION ON OOP**
*Frankfurt, Germany*
*49.61732852*

**October 18-22, 1993**
**C++ WORLD**
*Dallas, TX*
*212.274.9135*

**November 15-16,1993**
**COMPUTER WORLD EXPO**
*Frankfurt, Germany*
*800-225-4698*

**December 9-10,1993**
**DATABASE WORLD CLIENT/SERVER**
*Chicago, IL*
*508-470-3880/0526*

**April 25-28,1994**
**XWORLD'94**
*New York, NY*
*212.274.9135*

the method. Data you write to the Transcript should be identified, and should include some formatting such as tabs and carriage returns. Here is an example of an expression that would be inserted in the method of a class that understood the total message:

```
Transcript cr; show: 'Total = ' ,self total printString.
```

This expression prints the string 'Total =' concatenated with the string result of sending the total message to the receiver. The comma in the above expression is a message that returns the receiver concatenated with the argument, another string. Use it when you want to append a string. In this example, the result of the total message is an integer, so printString is used to obtain the string equivalent.

Use a global variable to control printing information to the Transcript, setting it to true or false from a workspace when you want to turn printing on or off. In this expression we use a global named Debug:

```
Debug ifTrue: [Transcript cr; show:'starting calculations...']
```

Instead of setting the global to a boolean, you can set the global to an integer that controls how much detail you print:

```
Debug > 4 ifTrue: [Transcript show:"detailed information"]
```

In Objectworks, you're not restricted to a single Transcript. If you would like to create customized transcripts to separate different types of messages, refer to the *Creating a transcript window* section on creating transcript windows in Chapter 21, "Text and text views," in the OBJECTWORKS SMALLTALK USER'S GUIDE.

### Menu Hooks for Inspectors

Printing a lot of information out to the Transcript can get rather tiresome. An attractive alternative is to open an inspector on key objects at strategic points in the code or, better yet, to provide an easy way for the developer to access an inspector. When you are creating new window applications, it's handy to include an inspect item in the window's menu during the initial development phase. This is a quick and easy way access the objects behind the window.

Inspect is implemented by Object, so you don't have to provide a new method if you're happy bringing up an inspector on the object that accepts responsibility for menu messages. If you do need to customize the inspect action from a window, provide a new message rather than overriding the inspect message. If you override inspect, your customized method, instead of the inherited method, will be executed by the system whenever the inspect message is sent to the object. Opening an inspector from an inspector, for example, uses the inspect message. If you want to inspect the selected item in a list directly from a menu, implement a new message called inspectSelectedItem and avoid overriding inspect.

### Object identity

Situations arise in which you need to compare two variables to see if they reference the same object. For example, you might be stepping through two similar sets of actions that involve a particular object. One works and the other doesn't, so you need to determine whether the two variables reference exactly the same object.

Object identity is determined with the = = message, which answers whether the receiver and the argument are *exactly* the same object. In contrast, the = message is used to determine object equality: It answers whether the receiver and the argument are equivalent:

```
#asdf == #asdf     "true: Symbols are unique."
'asdf' == 'asdf'   "false: Strings are not unique."
```

If the two objects are not in the same context (i.e., you have captured them in separate inspectors), you can assign one to a global variable and use the object identity message to determine equality.

```
GlobalOne := self name.      "in one inspector"
self name = = GlobalOne.     "in a different inspector"
```

Don't forget to remove global variables when you're through with them:

```
Smalltalk removeKey: #GlobalOne
```

Use standardized names, such as an unusual prefix, to identify temporary globals.

Older Smalltalk systems supported as hash as a means of uniquely identifying objects. In current Smalltalk systems, neither of these messages uniquely identify an object.

### Names

It is often a good idea to add a name or id field to an object strictly for debugging purposes, particularly when instances cannot be uniquely identified by their instance variables or when they are distinguished in obscure ways. If you're going to be dealing with multiple instances of a class, it may otherwise be hard to keep track of which object is which.

You also can specialize the method printOn: for your new classes. The printable representation can incorporate a name to help identify the object.

```
printOn: aStream
    "Add a printable representation of thereceiver to <aStream>.
    Use the fullName field to identify thereceiver."
    super printOn: aStream.
    aStream nextPutAll: ' on '.
    aStream nextPutAll: self fullName
```

A good printable representation can speed debugging, because it lets you quickly ascertain when two objects are equal or how they were created. However, be aware that assumptions in a specialized printOn: method might not be correct. For example, some instance variables might not have been initialized. If so, the previous method should be checked to see if the name were nil before printing it.

## WHERE AM I AND HOW DID I GET HERE?

An object encapsulates both behavior and data. In addition to gathering information about the data in your application, you may need to collect information about the dynamic state of your application. Two keys to understanding the dynamic state of your application are identifying where you are in the dynamic sequence of message sends and identifying how you got there.

We also present two alternate ways to access dynamic state: locating code of interest via user input and using key entry points.

**Identifying the Current Context**

When you need to identify the method you are executing, print an identification expression to the Transcript. The following prints the class and message name as it appears in the debugger's stack (e.g., Class(Superclass)>>methodName).For Objectworks:

```
"if it's not in a block"
Transcript show: thisContext printString; cr.

Debug ifTrue: ["use this expression in a block"
    Transcript show: thisContext sender home printString; cr].
```

For Smalltalk/V:

```
CurrentProcess walkbackOn: Transcript maxLevels: 1.
```

**Audible Feedback**

Another alternative to writing to the Transcript is to use sound to give audible feedback that a method has been executed. This is particularly useful in situations where the display system is not available. For example, in Smalltalk/V the GO file is processed before the display system is available. Insert these expressions to ring the bell. For Objectworks:

```
Screen default ringBell.
```

For Smalltalk/V:

```
Terminal bell.
```

**Catching It in the Act**

If you would like to examine code behind a specific action, but don't know where to find the method, you can interrupt it by typing the program interrupt while executing the code of interest. In Objectworks, the default program interrupt is <CTRL-C>. In Smalltalk/V, it is the platform interrupt key (<CTRL-BRK> under OS/2 and Windows, <command-.> on the Mac).

For example, if you want to know how the rubberbanding code works when drawing a line in a graphics editor:

1. Perform the appropriate action, such as holding down the left mouse button and dragging the cursor.
2. While you move the mouse, press the program interrupt.
3. A notifier appears that allows you to open a debugger and examine code in the stack. You can see flow of control in the debugger, and can examine method arguments and temporaries.

Timing can sometimes be a problem —for some operations you may need to try this several times until you catch it at the right place.

Sometimes a program interrupt can save you from a bad situation. If you make a simple change to your code and see a garbage collection cursor instead of what you expect, you may have created an infinite loop. The following is a typical example of a class method that inadvertently causes an infinite loop:

```
new
    ^self new initialize"this should be a call to super instead of to self"
```

In this method, the user intended to invoke the inherited method called new, but instead called the same method, resulting in an infinite loop.

If your application is in an infinite loop, you can interrupt it with a program interrupt. After interrupting the application, use the debugger to look at the stack and locate the error, fix the error and then either close the debugger and start again, or resume the execution from the debugger.

Be careful when you interrupt a method with a program interrupt. Instead of closing the notifier or debugger, you may need to resume or proceed from the debugger if you are in a loop that needs to finish execution to restore the state of the cursor, signal a semaphore, or complete some other clean-up activity.

**Alternative to Walkbacks and Notifiers**

You may not want to open a debugger and, instead, prefer some other way to view the context information. If you are debugging low-level code and are concerned that an interruption might leave the image in an unstable state, you can print out information about the current context as described below. It can also be useful if you are sending a beta release to customers or if you are working on an embedded application in which there is no access to a user interface. The following expression prints the execution stack on the Transcript. For Objectworks:

```
Transcript cr; show: (NotifierView shortStackFor: thisContext).
```

For Smalltalk/V:

```
CurrentProcess walkbackOn: Transcript maxLevels: 10.
```

You can also print this information to a file. For Objectworks:

```
| file |
file := 'errors' asFilename appendStream.
file cr; nextPutAll: (NotifierView shortStackFor: thisContext).
file close
```

For Smalltalk/V:

```
| file |
file := File pathName: 'errors'.
file setToEnd.
CurrentProcess walkbackOn: file maxLevels: 50.
file close
```

### Source Code for Blocks

Although the source code is not always available, the following expressions are sometimes helpful for examining the source code for blocks (Smalltalk/V) or BlockClosures or MethodContexts (Objectworks). For Objectworks:

```
aBlockClosure method getSource
aMethodContext sourceCode
```

For Smalltalk/V for OS/2:

```
aBlock homeContext method sourceString
```

### Decompiling a Method in Objectworks

If the source code for a method is unavailable, the Objectworks browser allows you to view a decompiled version of the method: The comments are gone, certain expressions are optimized, and the temporary variable names t1, t2, and so on are used in place of the original argument and temporary variable names.

Even when the source code is available, you can view the decompiled version of the method if you hold down the shift key when you select the method name in the Objectworks browser. This technique is useful for finding obscure bugs such as when literals have been unknowingly altered. Many programmers think that Smalltalk literals are immutable, and do not realize that they can be altered. The following example illustrates detection of an altered literal array.

A method initializes an instance variable to reference a literal array:

```
initialize
        arrayConstant := #(1 2 3 4)
```

The programmer intends this to be a constant, but later uses an expression such as the following to alter the array:

```
arrayConstant at: 1 put: 100
```

This alters the contents of the literal array in memory, so the original contents of the array are not restored even if the original initialize method is re-executed. You can check the contents of the literal array by decompiling any method that refers to it. After altering the array, the decompiled contents of the initialize method are:

```
arrayConstant := #(100 2 3 4)
```

If you recompile the method from the source, the original contents of the literal array are restored. This is a particularly nasty bug to locate, so be forewarned. To prevent this type of bug, some programmers provide accessing methods for important literals, and return a copy of the literal instead of the original. Because the original literal is never returned, inadvertent alterations are made only to the copy.

### Entry Points

Sometimes you just want to know how a window is opened or what happens when a menu item is invoked. Instead of interrupting it, sometimes it's easier to trace the action down from a few well-known entry points.

For example, the Objectworks Launcher lets you open browsers, workspaces, and other windows. The code behind this master menu is found in LauncherView and VisualWorks UIVisualLauncher class methods. Browse all implementors of '*enu*' to see menu initializations for other windows: select implementors from the VisualWorks Launcher Browse submenu or the ENVY Launcher ENVY>browseimplementors... alternative. The string '*enu*' matches selectors such as menu and fileListMenu regardless of the capitalization.

The file menu in Smalltalk/V contains items to open browsers, workspaces and other windows. The class ApplicationWindow supports the file menu, and contains entry points to tools. Browse the class to examine the methods that open windows.

## WHERE AM I GOING?

This section highlights techniques that allow you to temporarily halt or gain more control over the execution. Some techniques, such as slowing down the action in your application, are oriented towards graphical operations.

### Breakpoints

Although Smalltalk has a well-earned reputation for its debugging environment, current implementations place some restrictions on breakpoints. In Smalltalk/V, you can set breakpoints from a debugger. In Objectworks, you have to recompile a method and insert code to stop execution. Removing the code to stop execution also requires recompilation.

In both Smalltalk systems, one of the first debugging techniques you learn is to send the message halt to any object. When executed, it prompts you to open a debugger. In a debugger, you can execute expressions and inspect the current object, its instance variables, and any method temporaries. The message error: also prompts you open a debugger, and uses its argument in the title of the walkback or notifier. These expressions can be inserted in a method or executed in a workspace.

```
self halt.
self error: 'Invalid data during retrieval'
```

However, you quickly learn that this needs to be used with caution. If you place the expression inside a loop, a notifier appears each time the loop is executed. You can guard the expression if you know exactly when you want to break:

```
i > 10 ifTrue: [self halt]
```

Or you may choose to control the execution dynamically. For example, the following expression halts only if the shift key is pressed. For Objectworks:

```
InputState default shiftDown ifTrue: [self halt]
```

For Smalltalk/V:

```
(Notifier isKeyDown: VkShift) ifTrue: [self halt]
```

If the code is being executed from a controller method in Objectworks, you can use the simpler:

```
self sensor shiftDown ifTrue: [self halt]
```

If this interferes with other tests for the shift key, you can also test for the Meta, Option, Alt (if it isn't commandeered by your windowing system), or Ctrl keys. For more information, see the section on sensing input near the end of Chapter 18, "Application framework," of the USER'S GUIDE FOR OBJECTWORKS\SMALLTALK.

For Smalltalk/V, you can use platform-dependent keys with expressions such as the following. For Smalltalk/V for Windows, use:

```
(Notifier isKeyDown: VkControl) ifTrue: [self halt]
```

You can also gain control over the execution of non-primitive expressions executed in the context of a workspace, debugger, or inspector. For example, execute do it on the expression below, which sends the halt message to 3:

```
3 halt raisedTo: 2
```

In the debugger, step or skip through the messages until you get to the raisedTo: message and then send or hop. You can't step into a primitive, such as integer addition, from the debugger.

### Slowing Down the Action

Sometimes you don't actually want to stop the action; you just need to slow it down a little. For example, you're looking at code that draws a complicated figure with a loop and you want to see each line segment drawn, one at a time. You might use a delay in the loop For Objectworks:

```
Cursor wait showWhile: [(Delay forMilliseconds: 800) wait]
```

For Smalltalk/V for OS/2:

```
CursorManager wait changeFor: [DosLibrary sleep: 800]
```

In Objectworks, don't forget to send the wait message to the delay. You can create an instance of a Delay anytime you like, but it doesn't actually stop the action until the wait message is sent.

Or, you might choose to wait until a mouse button is clicked For Objectworks:

```
Cursor crossHair showWhile:
    [ScheduledControllers activeController sensor waitNoButton;
            waitClickButton]
```

This expression waits until all mouse buttons are up and then waits again until one is pressed. For Smalltalk/V:

```
CursorManager execute changeFor:
    [Notifier consumeInputUntil: [:event |
            event selector =#button1Down:].
    Notifier consumeInputUntil:[:event |
            event selector = #button1Up:]]
```

This expression waits until the left mouse button is pressed and then released.

The first expression makes sure you aren't in danger of running on through the whole expression just because the mouse button was still down from a previous operation such as a menu invocation.

Changing the cursor while the system is sleeping or waiting for a button press is a good visual reminder of your program's action. There are a number of other cursors available, and if you have multiple delays in a method, you can use different ones to give you feedback about the state of the execution.

A delay can also give you time to interrupt a method with a program interrupt if you so choose.

## HOW DO I GET OUT?

One of the best things about the Smalltalk environment is that you can change almost anything you like. One of the *worst* things about the Smalltalk environment is that you can change almost anything you like. If you happen to alter your environment in an undesirable way, you can also find yourself in big trouble.

Although you might be able get yourself out of a tight spot if you have enough time, skill, and patience, you may find that it's best to quit out of an image and recover desirable changes in a fresh image rather than to undo the damage.

### Quitting while You're Ahead

If your normal means of exiting is blocked, you can often exit by evaluating an expression. In Objectworks, the magic expression to gracefully shut down the image when all else has failed is:

```
ObjectMemory quit
```

or

```
ObjectMemory quitPrimitive
```

In pre-4.1 Objectworks, this message was sent to Smalltalk instead.

In Smalltalk/V, the expression is:

```
Smalltalk exit
```

If your image seems dead and you don't get any response from typing, first try the program interrupt and attempt the exit procedure again. If that doesn't work, then, for Objectworks, use the Emergency Evaluator to evaluate the exit expression:

1. Type <CTRL-SHIFT-C> to bring up the Emergency Evaluator.
2. Type the exit expression ObjectMemory quit.
3. Type <ESC> to evaluate the expression.

In Smalltalk/V for OS/2, use the WindowList provided by OS/2 to remove an unwanted process:

1. Type <CTRL-ESC> to bring up the WindowList.
2. Select the top-level Smalltalk/V Window or the Transcript.
3. Bring up the menu and select Close.

OS/2 also notices if a process is not responding to events and prompts you to exit the process.

In Smalltalk/V for Windows, you can use the WindowList provided by Windows to remove an unwanted task with the End Task button. A more reliable method is to type <CTRL-ALT-DEL>. The first <CTRL-ALT-DEL> allows you to exit the current process. Another <CTRL-ALT-DEL> allows you to reboot the machine.

After exiting, use the appropriate utilities to recover the changes you want to keep, being careful not to restore the method or methods that caused the crash.

### An Advanced Emergency Procedure for Objectworks

If you're feeling more adventuresome and know exactly what you did wrong, Objectworks allows you to recompile the offending method instead of quitting. For example, you insert a self halt in a critical method such as the otherwise empty controlInitialize1 method and quickly realize that you should have done this in a subclass when you see all the notifiers pop up. Since <CTRL-C>, the program interrupt key, doesn't help, you type <CTRL-SHIFT-C> to bring up the Emergency Evaluator and then evaluate the following expression to restore the original method:

```
Controller compile: 'controlInitialize  ^self' classified:
                  'basic control sequence'
```

Don't be concerned about making the method pretty or getting the protocol exactly correct; you can and should fix those details once your environment is back to normal again.

## OTHER DEBUGGING AIDS

Ways to debug problems are as varied as the bugs themselves, but the following tips include advice about general approaches to object-oriented debugging, techniques for graphical debugging and ways to add shortcuts to access frequently used debugging expressions.

### Isolate Debugging Code in a Subclass

Whenever possible, isolate your debugging code in a new subclass. You can copy methods from the superclass or override them to add debugging information. This is most useful when you are primarily interested in finding out how the current system works and your debugging activities are confined to halts and monitoring activities such as printing to the Transcript. If you are trying to find a genuine bug and functionally changing code, you have to remember to copy the changes back to the real class.

### Graphical Feedback

When you're debugging graphical applications, you need a lot of visual feedback. If your application is interactive, you might need to understand where the cursor is located and how to manipulate it.

In Objectworks, you can find out where the cursor is relative to the window with the expression:

```
ScheduledControllers activeController sensor cursorPoint
```

You can position the cursor explicitly with:

```
ScheduledControllers activeController sensor cursorPoint: aPoint
```

You can ask the user to interactively designate an area on the screen:

```
Rectangle fromUser
```

Indicate an area with a filled rectangle:

```
ScheduledControllers activeController view graphicsContext
    display Rectangle: (0@0 extent: 10@100)
```

In Smalltalk/V, there are a similar set of expressions. To find the location of the cursor in screen coordinates use:

```
Cursor sense
```

To translate to coordinates for a pane use:

```
Cursor sense mapScreenToClient: aPane
```

To set the location of the cursor relative to the screen origin:

```
Cursor offset: aPoint
```

You can ask the user to interactively designate an area on the screen:

```
Display rectangleFromUser
```

You can also indicate a screen area, in this case by filling the rectangle with a solid red color:

```
Display pen fill: Display rectangleFromUser color: ClrRed
```

### Magic Debugging Keys

If you find that you use certain debugging expressions frequently, you can modify your programming environment to add these expressions with function keys or keyboard equivalents.

For Objectworks, we use function keys to insert some of the debugging expressions mentioned previously.

The ParagraphEditor's initializeDispatchTable class method controls the binding of keys to actions. Rather than adding to this method, the following code creates a new method for the debugging bindings:

```
ParagraphEditorclass
initializeAdditionsToDispatchTable
    "Initialize additional keyboard dispatch keys."
    "ParagraphEditor initializeDispatchTable.
     ParagraphEditor initializeAdditionsToDispatchTable."
    Keyboard bindValue:#displayHaltKey: to: #F5.
    Keyboard bindValue:#displayGuardedHaltKey: to: #F6.
ParagraphEditor
displayHaltKey: aCharEvent
    "Replace the current text selection with a debugging statement—
initiated by #F5."
    self appendToSelection: 'self halt.\' withCRs.
displayGuardedHaltKey: aCharEvent
    "Replace the current textselection with a debugging statement—
initiated by #F6."
    self appendToSelection: 'InputState default shiftDown
        ifTrue:[self halt].\' withCRs.
```

After compiling those methods, be sure to execute

```
    ParagraphEditorinitializeDispatchTable.
    ParagraphEditorinitializeAdditionsToDispatchTable.
```

These bindings are valid only for windows created after initializing, so open a new Browser or Workspace to test the additions.

For Smalltalk/V, we use keyboard abbreviations. After typing an abbreviation, type Shift-Space to expand the abbreviation.

Execute the following code, customizing as appropriate:

```
    Smalltalk at: #Abbreviations put:Dictionary new.
    Abbreviations
        at: 'gh' put:'(Notifier isKeyDown: VkShift) ifTrue: [self halt].';
        at: 'tr' put:'CurrentProcess walkbackOn: Transcript maxLevels: 1.'
```

In Smalltalk/V for OS/2, add the following method:

```
        TextPane
    characterInput: aChar
        "Process a character typed by the user."
        | abbrevDict left right c scontinue newLine |
        abbrevDict := Smalltalk
            at:#Abbreviations
            ifAbsent: [^self basicCharacterInput:aChar].
        (aChar = Space and: [Notifier isShiftDown])
            ifTrue: [selfgetPMSelection.
                left := right := selEnd - 1.
                s := String new.
                [c := self charAt: left.
                    (continue := (c notNil and: [c isAlphaNumeric]))
                        ifTrue: [s := (String with: c), s].
                    continue]
                        whileTrue: [left := left - 1].
                new := abbrevDict at: s ifAbsent: [nil].
                new notNilifTrue: [self selectIndexFrom: left to:right].
                ^selfinsert: new].
        ^super characterInput: aChar
```

In Smalltalk/V for Windows, copy the text from the TextPane method characterInput: to a new method called basicCharacterInput:.

```
        TextPane
    basicCharacterInput: aChar
        "Private - the user typed aChar."
        self isGapSelection
            ifFalse: [selfhideSelection].
        newSelection := self replaceWithChar: aChar.
        modified :=true.
        self
            selectAfter: newSelection corner;
            makeSelectionVisible;
            displayChangesForCharInput;
            showSelection
```

Then, replace the original characterInput: method with the following:

```
        TextPane
    characterInput: aChar
        "Process a character typed by theuser."
        | abbrevDict left right c scontinue line new |
        abbrevDict := Smalltalk
                at: #Abbreviations
                ifAbsent: [^selfbasicCharacterInput: aChar].
        (aChar = Space and: [Notifier isKeyDown: VkShift])
            ifTrue:[left := right := selection corner x.
                line := textHolderlineAt: selection corner y
                s := String new.
                [c := line at: left.
                (continue := (c notNil and: [cisAlphaNumeric]))
                    ifTrue: [s := (String with: c),s]
                    ifFalse: [left := left + 1].
                (continue and: [left > 1])]
                    whileTrue: [left := left - 1].
                new :=abbrevDict at: s ifAbsent: [nil].
                new notNilifTrue:
                    [selection
                        selectBefore: left @ selection corner y;
                        selectTo: right @ selection corner y.
                    self replaceWithText: new.
                    selection selectAfter: left + new size @ selection corner y.
                    self forceSelectionOntoDisplay.
                    ^nil]].
        ^self basicCharacterInput: aChar
```

Be careful when entering this method in the browser, as mistakes will prevent subsequent character input from text panes, such as in the bottom pane of the browser.

## CAVEATS

Please note that the debugging techniques advocated in this article may violate normal programming guidelines. Some of the expressions use globals or "private" methods; others, like moving or warping the cursor, are expressly prohibited by user interface style guides. Use them judiciously.

## CONCLUSION

While this article has presented a collection of Smalltalk debugging techniques, it is impossible to describe the most efficient debugging strategy for any particular situation without knowing where the problem lies. Of course, if you knew where the bug was in the first place, you wouldn't need to debug it.

These debugging hints won't make you an expert overnight. Effective debugging requires creativity and experience and there are few shortcuts, but assembling an arsenal of debugging techniques can shorten development time and improve code quality. ■

**Acknowledgments**

We'd like to thank the following people, who provided problems, solutions, or otherwise helped debug the debugging paper: Ken Auer, Duane Campbell, Andrew Cornwall, Tom Hendley, Tom Heruska, Larry Jundt, Cary Laird, Mike Lucas, Pat Martin, Angie Multer, Kim Rochat, Brian Wilkerson.

*Roxie Rochat is Senior Technical Specialist in Advanced System Development and Process Instrumentation Technology at Fisher-Rosemount Systems Inc., 1712 Centre Creek Drive, Austin, TX 78754, 512.832.3583. She can be reached via email at rochat@fisher.com. Juanita Ewing is a senior staff member of Digtalk Professional Services, 921 SW Washington, Suite 312, Portland, OR 97205, 503.242.0725. She is a columnist for* THE SMALLTALK REPORT.

*...continued from page 6*

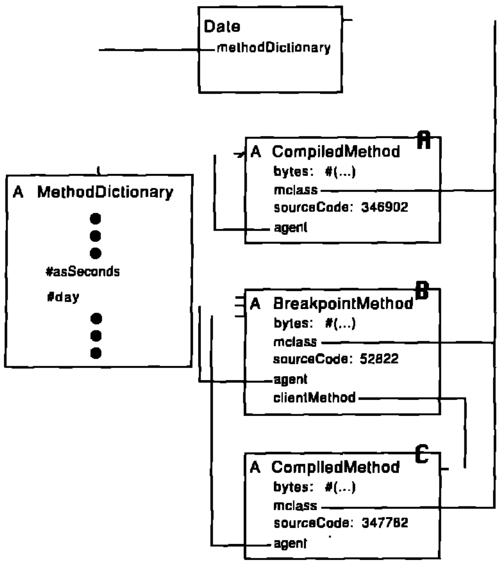


Figure 2. The relationship between CompiledMethods and the BreakpointMethods that represent them.

pointMethod 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.

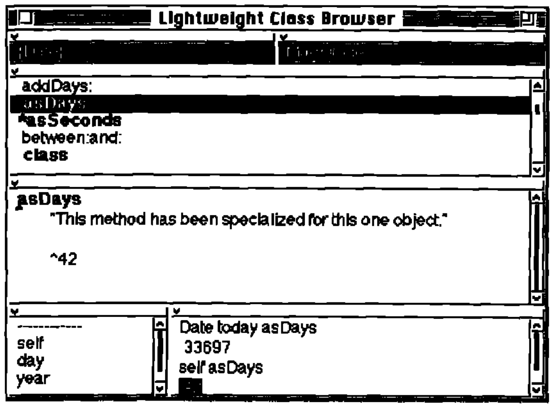
The new variable agent is needed to make CompiledMethods with breakpoints print out well. Every CompiledMethod has an instance variable called mclass, which refers to the class in whose method dictionary the CompiledMethod should be found. When CompiledMethods print themselves out, they look in their mclass to make sure they really are defined there ; if they aren't, 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.

In 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



```
addDays:
asDays:
asSeconds
between:and:
class
```

```
asDays
    "This method has been specialized for this one object."

    ^42
```

```
----------
self
day
year
```

```
Date today asDays
 33697
self asDays
```

Figure 3. The lightweight class browser.

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 Breakpoint-Methods.

## 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 is 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.

LightweightClassBrowser is a subclass of Browser for looking at lightweight classes. As shown in Figure 3, the Lightweight-ClassBrowser 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 is 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 browse-Lightweight 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 to 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. Beck, K. Instance-specific behavior, part I, THE SMALLTALK REPORT 2(6), 1992.
2. Beck, K. Instance-specific behavior, part II, THE SMALLTALK REPORT 2(7), 1992.
3. Hinkle, B. and R. E. Johnson. Taking exception to Smalltalk, part 1, THE SMALLTALK REPORT 2(3), 1992.
4. Hinkle, B., and R. E. Johnson. Taking exception to Smalltalk, part 2, THE SMALLTALK REPORT, (2)4, 1993.
5. Goldberg, A., and D. Robson. SMALLTALK-80: THE LANGUAGE AND ITS IMPLEMENTATION, Addison-Wesley, Reading, MA, 1983.
6. 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.

*Bob Hinkle, Vicki Jones, and Ralph E. Johnson are affiliated with the Department of Computer Science at University of Illinois at Urbana-Champaign. Bob Hinkle is supported by a fellowship from the Fannie and John Hertz Foundation. He can be reached via email at r-hinkle@uiuc.edu. Vicki Jones and Ralph Johnson can be reached via email at {vjones, johnson}@cs.uiuc.edu.*

# Highlights

## Excerpts from industry publications

### COBOL TO OOP

What would you say if your boss ordered you to transform 60 mainframe programmers into object-oriented programmers in one year? Most likely, "You're joking, right?" Believe it or not, in the past year American Management Systems (AMS) of Arlington, Va., has transformed over 60 COBOL programmers into Smalltalk GUI programmers. They didn't raid the staff of an OOP tools firm, and they didn't rely heavily on external consultants. But they did perform a major paradigm shift on the minority of their staff. . . The secrets of their success included: Boot camp: All programmers went through development tool training and object-oriented design training. The majority participated in a one- to eight-week apprenticeship program, where they worked side by side with object-oriented pros. The process was supportive and orderly—at no point did programmers feel they were floundering. Teamwork: AMS brought in OOP design dn programming experts to "mind-meld" with their COBOL programmers. The experts designed the application architectures and classes; the novices handled the specialized processing and application logic. The OOP novices with GUI design expertise did the screen layout. The managers performed function-point analysis to glean new project-estimation metrics. AMS effectively used consultants to jump-start their efforts, without paying a fortune. Today they have a core team of strong OOP technicians in-house. . .

*Bringing object-oriented technology to the masses, Christine Comaford, PC Week, 2/27/93*

### THEY SAY WE HAVE A REVOLUTION

We are currently in the middle of a revolution in the Smalltalk world. Back in the old days the only objects that came with any language were simple data structures, enough metaobjects to write the system itself, and support for rudimentary graphics and user interfaces. Everyone who used an object language was in the business, by necessity, of creating fundamentally new kinds of objects all the time. This limited users to those who were capable of such invention,a nd limited the productivity of those users because writing new kinds of things is so much harder than reusing existing frameworks. A consensus has grown recently that the time has come to stop focusing exclusively on creating objects and start supporting people who only want to use or elaborate on things that already exist. Several factors contributed to this shift: The market of wizards creating new frameworks from scratch was getting saturated. The economics of growth dictates a search for new kinds of customers. The pace of innovation in user interfaces slowed, with the major windowing systems settling on roughly the same set of components. This allowed the Smalltalk vendors to stop spending so much energy doing the entire user interface without help from the operating system. Enough objects had been created that is was possible to imagine someone writing an application and not having to create new kinds of objects. The factors that used to single out Smalltalk—a bundled class library and an interactive programming environment—were no longer unique. Smalltalk had to move on or get trampled by the Borland C++'s of the world. . .

*Whole lotta Smalltalk, Kent Beck, Object Magazine, 3-4/93*

### CORBA

About 60 companies are creating CORBA implementations, according to the Object Management Group. But only DEC and HyperDesk, Westborough, Mass., with its Distributed Object Management System, are shipping CORBA 1.1 products. . . HP's implementation, to be called HP Distributed Smalltalk, is a set of Smalltalk classes for use with VisualWorks, a Smalltalk development environment from ParcPlace Systems. . .

*HP Tool Showcases Key Object Spec, Dan Richman, Open Systems Today, 2/15/93*

### OBJECT SQL

DBMS: Are you planning an object-oriented language? Or do you recommend one?

[R&D section manager for HP's Database Lab, and second chairman of the SQL Access Group: John R. Robertson]:The real issue is moving into that paradigm. Yes, we should have standards, we should have a common language. I don't think C++ is necessarily the right language. By the time you get into object systems you probably want to be having 4 GLs that are going to take care of it for your. We should've learned that lesson by now. We are not making an object-oriented language. We have an object-interactive language, which really operates at the command level. We're working with third parties who are in the 4GL business. The Object Management Group working group seems to be migrating toward having a common command set, which is OSQL [Object SQL]. I don't think it matters much whether you express that through C++ or Smalltalk. The real issue is that you want your object model to let you move your methods out of your application and put them into the database where you can reuse them. This is how database technology will mature.

*Hewlett-Packard's Relational/Object Paradigm, Peggy Watt and Joe Celko, DBMS, 2/93*

## PRODUCT ANNOUNCEMENTS

**Product Announcements are not reviews. They are abstracted from press releases provided by vendors, and no endorsement is implied. Vendors interested in being included in this feature should send press releases to our editorial offfices, Product Announcements Dept., 91 Second Ave., Ottawa, Ontario K1S 2H4, Canada.**

### SERVIO TO SUPPORT GEMSTONE ODBMS, GEODE DEVELOPMENT ENVIRONMENT ON WINDOWS NT.

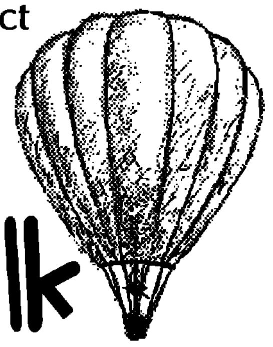
Servio Corporation has announced that it will provide support for its full range of products on Microsoft Corporation's Windows NT operating system.

GemStone and GeODE for Windows NT are scheduled for production shipment beginning in early 1994. They are currently available for most leading UNIX-based platforms including Sequent Symmetry 2000, SUN SPARC, RS6000, and HP9000, GemStone release 3.2 and GeODE release 2.0. GemStone is also available for DEC VAX/VMS. GemStone data can be accessed from most client environments including UNIX, Windows, OS/2, and Macintosh.

Servio Corporation develops and markets the GemStone object database management system, which incorporates the GeODE code-free visual development environment for rapidly building and deploying end-user database applications. Servio supports its products with consulting on-site technical support and educational services that enable customers to implement mission-critical object-based solutions.

*Servio Corp., 2085 Hamilton Ave., Ste. 200, San Jose, CA 95125, 408.879.6200 (v), 408.869.0422 (f)*

The first multi-dialect
**Smalltalk**
developers
conference



DevTalk

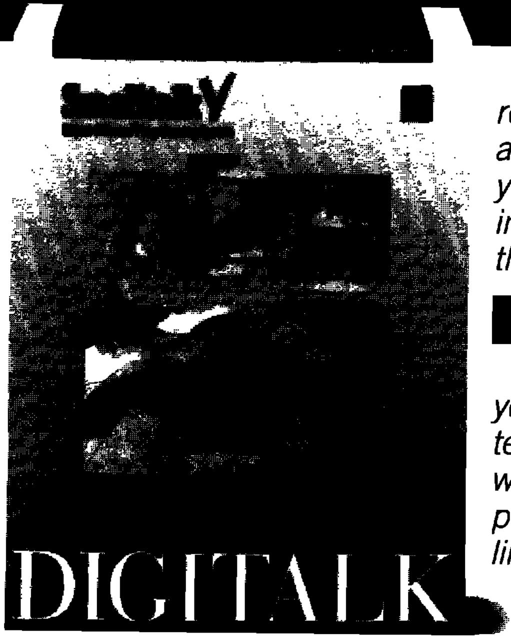
**August 20-21, Glendale, California.**

Presentations · Panels · Tutorials · Technical sessions · Exhibits · Books & Magazines.

Digitalk (*Smalltalk/V*) · Servio (*GemStone*) · Easel (*ENFIN*) · Quasar (*SmalltalkAgents*) · ObjectShare (*WindowBuilder*) · ParcPlace (*ObjectWorks*) · many others.

Only $250 if you register before July 30! $300 after. For more information and a registration form, contact Monica at (tel) 213-257-5670, (fax) 213-259-0430, or (Compuserve) 72330,1236.

# THE TOP NAME IN TRAINING IS ON THE BOTTOM OF THE BOX.



*Where can you find the best in object-oriented training?*

*The same place you found the best in object-oriented products. At Digitalk, the creator of Smalltalk/V.*

*Whether you're launching a pilot project, modernizing legacy code, or developing a large scale application, nobody else can contribute such inside expertise. Training, design, consulting, prototyping, mentoring, custom engineering, and project planning. For Windows, OS/2 or Macintosh. Digitalk does it all.*

**ONE-STOP SHOPPING.**

*Only Digitalk offers you a complete solution. Including award-winning products, proven training and our arsenal of consulting services.*

*Which you can benefit from on-site, or at our training facilities in Oregon. Either way, you'll learn from a staff that literally wrote the book on object-oriented design (the internationally respected "Designing Object Oriented Software").*

*We know objects and Smalltalk/V inside out, because we've been developing real-world applications for years.*

*The result? You'll absorb the tips, techniques and strategies that immediately boost your productivity. You'll reduce your learning curve, and you'll meet or exceed your project expectations. All in a time frame you may now think impossible.*

**IMMEDIATE RESULTS.**

*Digitalk's training gives you practical information and techniques you can put to work immediately on your project. Just ask our clients like IBM, Bank of America, Progressive Insurance, Puget Power & Light, U.S. Sprint, plus many others. And Digitalk is one of only eight companies in IBM's International Alliance for AD/Cycle—IBM's software development strategy for the 1990's. For a full description and schedule of classes, call (800) 888-6892 x411.*

*Let the people who put the power in Smalltalk/V, help you get the most power out of it.*

**100% PURE OBJECT TRAINING.** DIGITALK