THE **Smalltalk** REPORT
PROVIDING SOLUTIONS TO THE SMALLTALK COMMUNITY

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

SIGS Publications Advisory Board
Tom Atwood, *Object Design*
François Bancilhon, *O₂ Technology*
Grady Booch, *Rational*
George Bosworth, *ParcPlace-Digitalk*
Jesse Michael Chonoles, *Lockheed Martin ACC*
Stuart Frost, *SELECT Software*
Adele Goldberg, *ParcPlace-Digitalk*
Thomas Keffer, *Rogue Wave Software*
R. Jordan Kriendler, *IBM Consulting Group*
Thomas Love, *Consultant*
Bertrand Meyer, *ISE*
Meilir Page-Jones, *Wayland Systems*
Cliff Reeves, *IBM*
Bjarne Stroustrup, *AT&T Bell Labs*
Dave Thomas, *Object Technology International*

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

Columnists
Jay Almarode, *GemStone Systems Inc.*
Kent Beck, *First Class Software*
Juanita Ewing, *ParcPlace-Digitalk*
Greg Hendley, *Knowledge Systems Corp.*
Tim Howard, *FH Protocol, Inc.*
Alan Knight, *The Object People*
William Kohl, *RothWell International*
Mark Lorenz, *Hatteras Software, Inc.*
Eric Smith, *Knowledge Systems Corp.*
Rebecca Wirfs-Brock, *ParcPlace-Digitalk*

SIGS Publications Group, Inc.
Richard P. Friedman, Founder, President, and CEO
Hal Avery, Group Publisher

Editorial/Production
Kristina Joukhadar, Editorial Director
Elisa Varian, Production Manager
Andrea Cammarata, Art Director
Elizabeth A. Upp, Associate Managing Editor
Margaret Conti, Advertising Production Coordinator
Shannon Smith, Editorial Production Assistant

Circulation
Bruce Shriver, Jr., Circulation Director
Lawrence E. Hoffer, Marketing Manager

Advertising/Marketing
Gary Portie, Advertising Manager, East Coast/Canada/Europe
Michael W. Peck, Advertising Representative
Kristine Viksnins, West Coast Exhibit Sales
Sarah Olszewski, East Coast Exhibit Sales
212.242.7447 (v), 212.242.7574 (f)
Diane Fuller & Associates, Sales Representative, West Coast
408.255.2991 (v), 408.255.2992 (f)
Sarah Hamilton, Director of Promotions and Research
Wendy Dinbokowitz, Promotions Manager for Magazines

Administration
Margherita R. Monck, General Manager
David Chatterpaul, Senior Accounting Manager
Bibi Budhram, Accounts Payable

Publishers of JOURNAL OF OBJECT-ORIENTED PROGRAMMING, OBJECT MAGAZINE, C++ REPORT, THE SMALLTALK REPORT, THE X JOURNAL, REPORT ON OBJECT ANALYSIS & DESIGN, OBJECTS EXPERT (UK), and OBJEKT SPEKTRUM (GERMANY)

Cover illustration by Dave Cutler/SIS

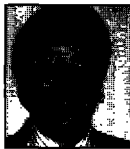
Table of Contents

October 1995 Vol 5 No 2

Features

Coverage analysis in Smalltalk 4
Mark Murphy
Coverage analysis greatly benefits quality assurance efforts on Smalltalk projects by identifying areas that have been inadequately tested. An implementation in Visual Smalltalk is presented.

Columns



Deep in the Heart of Smalltalk 9
Breakpoints revisited
Bob Hinkle and Ralph E. Johnson
These new breakpoints and lightweight classes narrow the focus of debugging, allowing programmers to investigate close to suspected bugs without wading through extraneous information.

Managing Objects 16
Exploiting stability
Jan Steinman and Barbara Yates
Successful team Smalltalk demands that synchronization and coordination take place during periods of maximum stability. The techniques presented help you detect and make the best use of such stability.

Project Practicalities 19
Improving your designs
by Mark Lorenz
A metrics process such as this one, which measures inheritance, collaboration, encapsulation, and design techniques, will help you build higher-quality systems using OT.

Getting Real 22
Class versioning and instance migration
Jay Almarode
In the dynamic Smalltalk environment, you need a strategy for when class modification causes structural changes to instances.

comp.lang.smalltalk 24
Hardware
Alan Knight
Could we build a Smalltalk processor that runs Smalltalk as fast as C runs on a conventional CPU? More importantly, should we?

Departments

Editors' Corner 2
Interview 28
Recruitment 30

The Smalltalk Report (ISSN# 1056-7976) is published 9 times a year, monthly except in Mar–Apr, July–Aug, and Nov–Dec. Published by SIGS Publications Inc., 71 West 23rd St., 3rd Floor, New York, NY 10010. © Copyright 1995 by SIGS Publications. 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 publisher. Second Class Postage Pending at NY, NY and additional Mailing offices. Canada Post International Publications Mail Product Sales Agreement No. 290386.

Individual Subscription rates 1 year (9 issues): domestic $89; Mexico and Canada $114, Foreign $129; Institutional/Library rates: domestic $199, Canada & Mexico $224, Foreign $239. To submit articles, please send electronic files on disk to the Editors at 885 Meadowlands Drive #509, Ottawa, Ontario K2C 3N2, Canada, or via Internet to streport@objectpeople.on.ca. Preferred formats for figures are Mac or DOS EPS, TIF, or GIF formats. Always send a paper copy of your manuscript, including camera-ready copies of your figures (laser output is fine).

POSTMASTER: Send domestic address changes and subscription orders to: The Smalltalk Report, P.O. Box 5050, Brentwood, TN 37024-5050. For service on current domestic subscriptions call 1.800.361.1279 or fax 615.370.4845. Email: subscriptions@sigs.com. For foreign subscription orders and inquiries phone +44(0)1858.435302. PRINTED IN THE UNITED STATES.

Editors' Corner

John Pugh

Paul White

As school session has begun again around North America, one question we often get asked is "why aren't more schools teaching Smalltalk?" To us, it has been a point of frustration that there has not been a great increase in the number of schools using Smalltalk somewhere in their curriculum. The university with which we are affiliated, of course, was (and continues to be!) a pioneer in the use of Smalltalk. At Carleton, we introduce students to Smalltalk during their first week in the program, and use it pervasively throughout. Smalltalk is used when teaching programming techniques, data structures, programming language concepts, and a variety of advanced topics such as computer animation and expert systems. Object-oriented software engineering principles are taught ahead of structured techniques, although both are covered in detail (as well as a number of courses that utilize more traditional CS languages such as C and UNIX).

We think the end products produced—our graduates—are as strong or stronger as those from any other school. And this strength is more than just with Smalltalk programming—they are very strong software engineers. The feedback from employers has been excellent. When talking to one employer, their comment was "these students understand that software development is not an individual thing; that software generated must be integrated with other software." This was interesting because the students were actually co-op students carrying out C development. It seems the principles of data and procedural abstraction, which are introduced immediately when using Smalltalk, are taken with the students as they move forward into other application areas. The old adage of "what is learned first is what is always remembered" is true for software engineers.

So, why aren't computer science departments rushing to jump on the Smalltalk bandwagon? It seems, unfortunately, that they believe it's too much work. A program as radical as the one we have implemented, for example, does have a fundamental impact on everything done within the program, and requires a commitment from faculty that is difficult to attain. An interesting case was a person teaching analysis of algorithms at our school, who felt sheltered from this "object nonsense"—but when students submitted solutions for his first assignment, the pseudocode was all Smalltalk! We recognize that it isn't realistic for a single champion of Smalltalk in a department to implement the changes necessary; it really requires the will of a significant number of faculty to make it happen, and that, unfortunately, does not appear to be happening in most cases.

It seems only when funding starts to be impacted, or enough pressure from industry gets exerted, that [CS] departments begin to change.

Another reason for this lack of movement to objects is that it is perceived by many of the "ivory tower" crowd to be just another passing craze. Unlike the move to structured programming, which was really led by academics rather than industry, the move to object-oriented programming has very much been driven by industry. On a positive note, we are now seeing some colleges getting into the act, and in a hurry. We had the pleasure this past summer of spending some time with the entire faculty of one community college that has decided to turn its complete CS curriculum upside down and become an object-oriented school. Clearly this was a painful thing to do but, when asked, the faculty answered frankly that "their advisory board, made up entirely of members of local industry (who hire their students) said failing to do so would be doing their students a disservice."

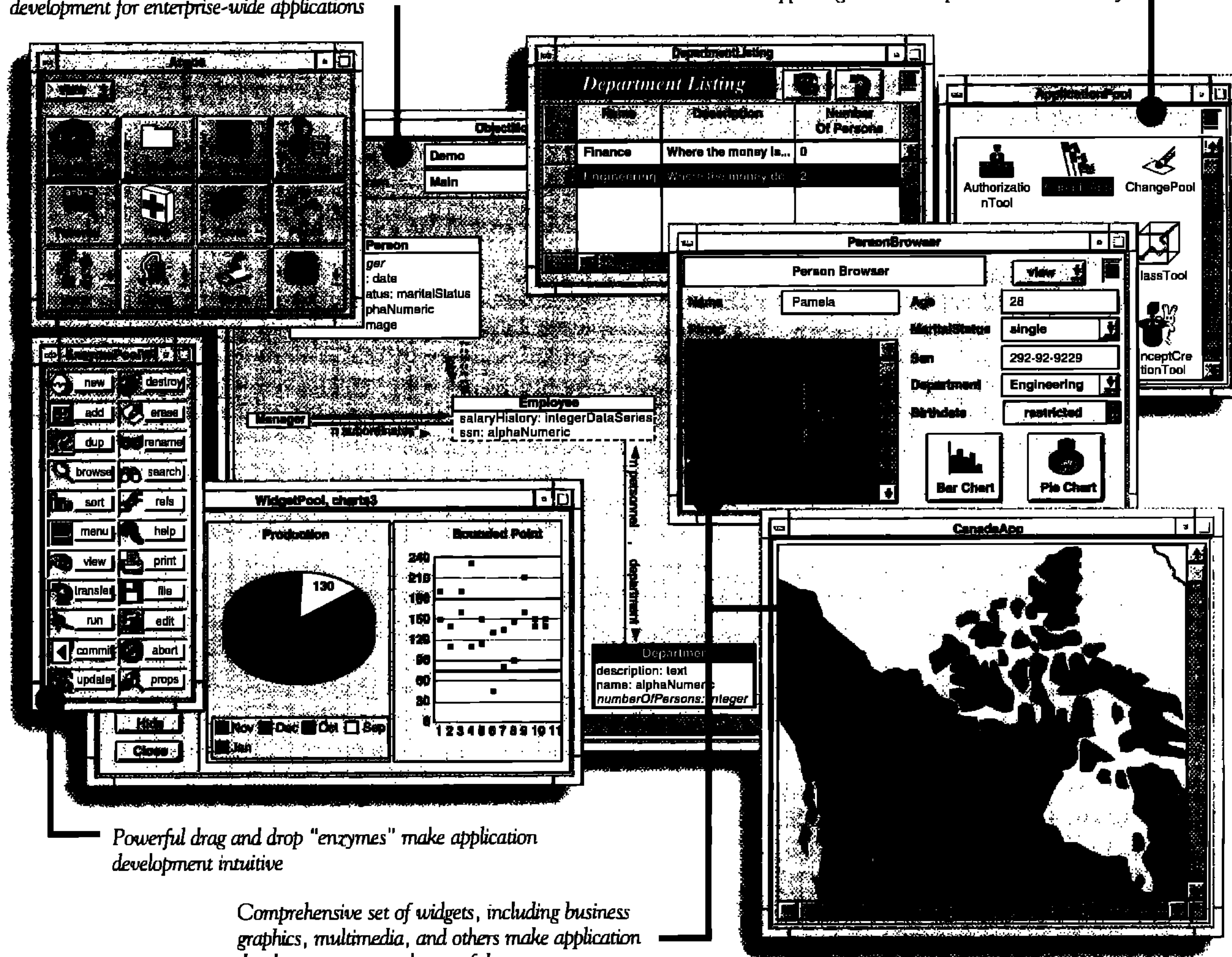
What is needed, of course, is for other "industry advisory boards" to shake up many other departments. It seems only when funding starts to be impacted, or enough pressure from industry gets exerted, that departments begin to change. Well, we suggest that you in industry let your alma maters know that you think the software engineers they are producing should be better equipped for the new software field of today.

And for those of you in departments who are trying to make these changes, some words of advice. First, make sure that both the people who teach objects in general, and those who teach Smalltalk specifically, understand the language. Put in the wrong hands, it is doomed to fail. Second, make sure that you make arrangements for access to the software ahead of time.

continued on page 32

Introducing Argos

The only end-to-end object development and deployment solution



VERSANT Argos™ is the only application development environment (ADE) that makes it easy to build and deploy powerful, enterprise-wide object applications. Easy because Argos features an embedded modeling tool and Smalltalk code generation that ensure synchronization between your models and applications. Powerful because Argos supports full traceability and workgroup development through a shared repository.

Argos automatically generates multi-user database applications that run on the industry-leading VERSANT ODBMS. Argos deals with critical issues such as locking and concurrency control transparently. And only Argos is packaged as a completely visual ADE built on ParcPlace VisualWorks®.

Leading organizations — in industries from telecommunications to finance — are using Argos to deliver business-critical applications. Find out how Argos can help you deliver your critical applications in weeks, instead of years.

Contact us today at 1-800-VERSANT, ext. 415 or via e-mail at info@versant.com

VERSANT
The Database For Objects™

1380 Willow Road • Menlo Park, CA 94025 • (415) 329-7500

©1994 by Versant Object Technology. VERSANT, VERSANT Argos and The Database For Objects are trademarks of Versant Object Technology Corporation. All other company names and logos are registered trademarks of the individual companies.

Coverage analysis in Smalltalk

Mark L. Murphy

HOW MUCH IS ENOUGH?

Quality assurance, unfortunately, is largely a guessing game. You know when programming is complete on a project, because all the requirements have been met. However, the standard against which quality assurance is held is, "Are there any bugs left?" which is impossible to know. You could test until the end of time and still miss bugs. Every project, therefore, makes a decision as to how much testing is enough.

Project managers are charged with ensuring that the testing is done, and done properly. It is not always easy to tell, however. Suppose you ask to see the unit test plan for a 24-class subsystem, and the developer hands you a 7-page document outlining a dozen tests. It is quite likely, just by looking at it, that the test suite is insufficient to really test out all the subsystem's functionality—it is simply too short. Suppose, however, that the test plan were 70 pages, or 700. How do you know if the tests really do thoroughly exercise the subsystem? A 700-page test plan may seem to be good based on size alone, yet might miss entire classes, if the suite is really bad. You just do not know for sure.

What we need is an objective measurement and set of criteria for determining test completeness. *Coverage analysis* is one such metric. This article will describe what coverage analysis means and how one can use it in practice. It includes an overview of some classes for measuring method coverage in Visual Smalltalk 3.0.1 (VST).

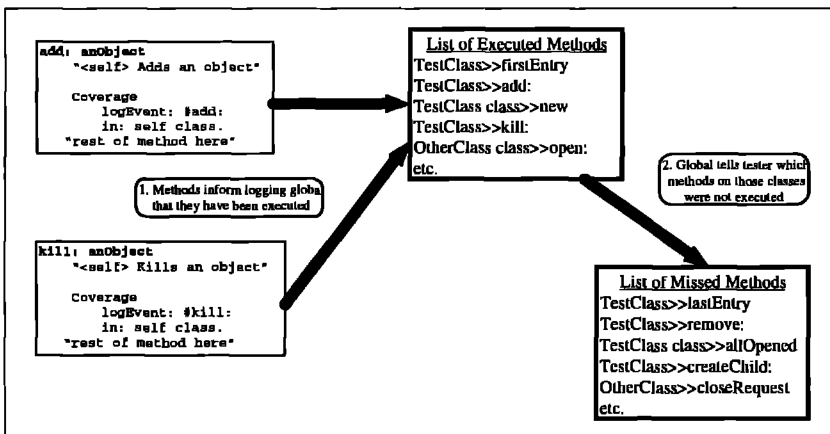


Figure 1. Method coverage in action.

ENTER COVERAGE ANALYSIS

Coverage analysis involves "teaching" the subsystem to track what portions of it a test suite executes. For example, each method might note that a test executed it before evaluating the body of the method itself (see Fig. 1). One can run a test suite, collect coverage tracking information, and determine specifically what the test executed and what it did not.

Now you have real-world information to determine how thorough a test suite is. The spots that were missed represent areas that the suite cannot audit. If they were not tested by hand in some other way, they have not been tested at all, and may contain bugs. You can even express coverage data in metric form (e.g., "we covered 87% of the subsystem's methods") for use in overall project benchmarking.

COVERING WHAT? AND HOW?

A generic coverage analysis tool will not know specifics about an application. Hence, it cannot say: "You missed testing the X-Base file filter."

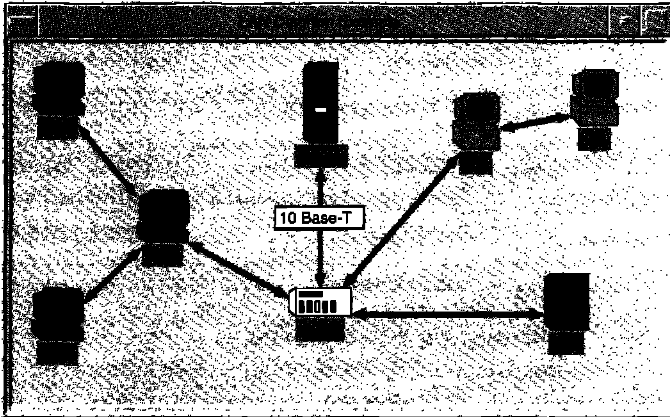
All the analyzer can do is report on abstract coverage metrics. For example, method coverage asks the question, "Has every method been executed at least once?" If you've never tested a method, you cannot know if it works!

Researchers and practitioners have identified several types of useful coverage metrics, each asking a different question. These include:

- **Statement**: Has each line of code been executed at least once?
- **Branch**: Has every logical branch (e.g., ifTrue: and ifFalse:) been tried in both the true and false directions?
- **Loop**: Has every loop (e.g., do: on a collection) been tried with 0, 1, and several passes through the loop? Zero passes means that the loop body (e.g., the block passed to do:) is never evaluated.
- **Path**: Has every logical path through a method been tried? Figure 2 shows a sample Smalltalk method and the four paths that an application could take. Branch coverage would be satisfied by testing only two of the paths: A and D.

Now it's Easy to Build Interactive Diagrams

Quickly create advanced interfaces that convey information better than lists... with DDF

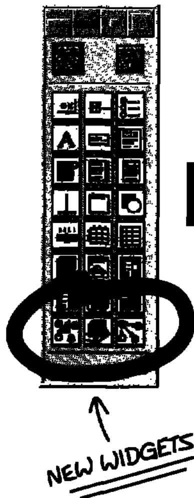


Example interface built with the Dynamic Diagram Framework

DDF™ is an easy-to-use tool that dramatically reduces the time needed to build interfaces:

- makes building diagrams simple
- provides a new VisualWorks widget
- pre-configured for immediate use
- written completely in Smalltalk
- refineable and extendable
- includes ARS's Parcels & Structured Graphics for building "dynamic" nodes

DDF - Dynamic Diagram Framework



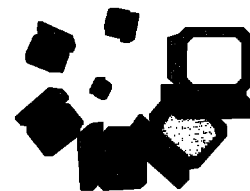
With DDF™ you can quickly and easily ...

- create customized node icons with shapes
- add and remove nodes from a diagram
- connect and disconnect nodes within a diagram
- customize lines and line decorations
- select and move nodes
- format diagrams and hide nodes
- print and store diagrams
- dynamically update diagrams

P&SG - Parcels & Structured Graphics

Parcels & Structured Graphics (P&SG™)

- high precision 2-d object-oriented graphics for VisualWorks
- structured graphic shape objects
- drag-and-drop with parcels
- shapes recognize "hot-spots"
- provides 2 new VisualWorks widgets



Shapes can be rotated, translated, scaled and combined to form new shapes.

Call (800) 260-2772 today to order or e-mail info@arscorp.com for more information. Ask for a free copy of the white-paper "Building Diagram-Based Applications with DDF"

Also Available: **MI - Multiple Inheritance**

Applied Reasoning Systems Corporation (ARS) is an innovative developer of high quality Smalltalk development tools, application frameworks, intelligent software systems, and related services that provide advanced solutions to complex problems.

Smalltalk Products • Consulting • Education • Mentoring

Applied Reasoning Systems

Phone: (919) 781-7997 • E-mail: info@arscorp.com

ANALYZING THE RESULTS

From a quality-assurance perspective, coverage analysis will provide a list of missed pieces of code, along with an overall percentage of code coverage. Developers must decide on an acceptable coverage level for a project. If a unit test for a subsystem fails to meet that metric, developers can add more tests based on the list of missed coverage spots.

Note, however, that some coverage "misses" may actually be expected. For example, perhaps there is some code that specifically handles the exception raised when a database server is not responding. Testing that code in a live situation means bringing down the server, which database administrators typically dislike. Hence, the test suite intentionally might skip a test for that exception handler, but the coverage analysis will still report that the suite missed it. Hence, a sub-100% coverage value may still be acceptable.

What you really want is 100% practical coverage, where testers justify each missed method. Note that those methods still must be tested somehow, even if it is by hand. Since you cannot continually re-test that method (regression testing), make sure it is right the first time!

METHOD COVERAGE IN VST

If you want to employ method coverage in your Smalltalk application, you have three main choices:

1. Hand-code messages to a coverage-logging routine, as was shown in Figure 1—but this is time consuming.
2. Use the profiler that comes with the Smalltalk development environment.
3. Use or construct custom classes specifically for doing method coverage analysis in an automated fashion.

Using a profiler

Many Smalltalk implementations come with profiling tools used for measuring the performance of pieces of Smalltalk code. One could, in principle, use them for method coverage analysis as well, since they watch over what a test executes. The TimeProfiler that comes with VisualWorks would not work in this case, because it uses a statistical sampling technique that might miss methods.

The profiler that comes with VST, however, does catch each and every message an application sends. Why, then, build another set of classes for method coverage? There are two reasons:

1. The VST profiler watches all methods in the image. Even the simplest test generates dozens to thousands of messages, most of which are for Smalltalk kernel classes. It would be difficult to determine the use of application classes with all this "noise."
2. The profiler does not report the methods the test missed, only those that it executed (and how long they took). Developers would have to manually cross-reference against the class to figure out which ones the test missed.

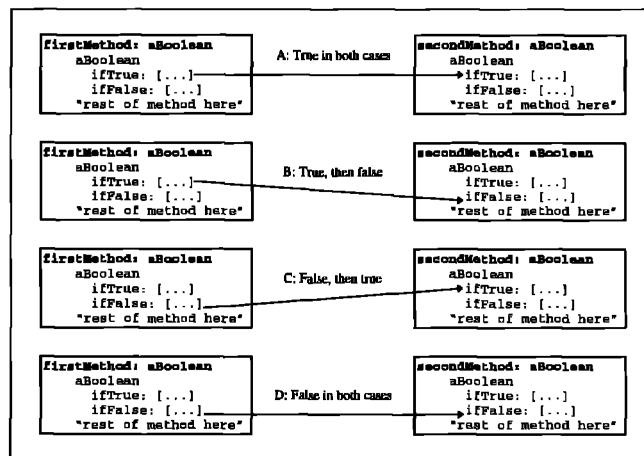


Figure 2. Paths to be covered.

Technique: Method wrapping

What we want to do is create a mechanism of "instrumenting" methods: adding code that does not change their original behavior but adds in new functionality. For coverage analysis, the instrumentation will simply inform some Smalltalk global that this method has been invoked. That global will have to track the called methods and provide coverage results on demand when the test is done, as shown in Figure 1.

The term for this instrumentation technique is "method wrapping." There are two ways one can wrap a method. One is simply to add new Smalltalk code to the source of the existing method and recompile it. This will work in many situations, but not all. Primitive methods cannot be wrapped this way, because Smalltalk code and a primitive call cannot coexist. It also requires one to parse Smalltalk methods, which is a nuisance.

Another approach, moving the existing method, is used both by the TimeProfiler that comes with VisualWorks Advanced Tools and the classes in this article. The CompiledMethod to be wrapped is moved in the class' method dictionary from its original selector to a new, unused selector. Then, a new CompiledMethod is created and installed under the original selector. This new

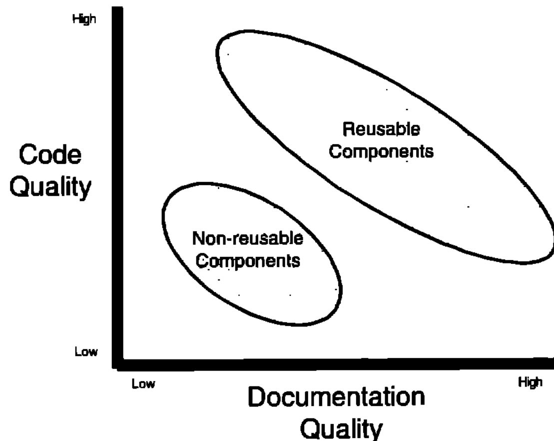
Table 1. Method dictionary before instrumentation.

Selector	Method
log:forClass:	**log: aSymbol forClass: a Class** "Execute real code here"

Table 2. Method dictionary after instrumentation.

Selector	Method
real_log:forClass:	**log: aSymbol forClass: a Class** "Execute real code here"
log:forClass:	**log: p1 forClass: p2** "Execute instrumentation code here" ^self real_log: p1 forClass: p2

Reuse Depends on Quality Documentation



Maximize Reuse

Many things are needed to have reusable software. However, if developers cannot understand available software, it is not going to be reused.

Reusable software requires readily available, high quality documentation.

And the easiest way for Smalltalk developers to get quality documentation is with Synopsis. Install it and see immediate results!

Features of Synopsis

- Documents Classes Automatically
- Builds Class or Subsystem Encyclopedias
- Moves Documentation to Word Processors
- Packages Encyclopedias as Help Files

Synopsis Software

8912 Oxbridge Court, Suite 300, Raleigh NC 27613
Phone 919-847-2221 Fax 919-676-7501

Products

Synopsis for IBM Smalltalk $295 Team $395 New!
Synopsis for Smalltalk/V and Team/V $295
Synopsis for ENVY/Developer for Smalltalk/V $395

method executes the instrumentation code, then sends a message, using the temporary selector, to invoke the original behavior. Since the original CompiledMethod remains unchanged, any type of method can be wrapped. Tables 1 and 2 show a fragment of a class' method dictionary, both before and after instrumentation.

In either case, to "unwrap" the method one simply restores the original compiled method under its original selector. While a method is wrapped, it will perform the added code (e.g., log method execution to a global) in addition to its old behavior. Once you unwrap it, everything returns to normal.

The VST implementation of coverage analysis via method wrapping for this article involves three classes:

- SRInstrumentedMethod—can wrap and unwrap a specified method.
- SRCoveredMethod—a subclass of SRInstrumentedMethod that wraps methods with coverage-logging statements.
- SRCoverageMonitor—logs coverage events created by the SRCoveredMethod instances.

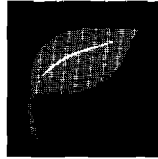
These classes are available on the Internet at http.//www.evro.com/STReport/Oct 95.htm.

Using the monitor

The only class that developers need to use directly is SRCoverageMonitor. The following steps describe how to start and stop coverage and get results:

1. Create a new instance of a SRCoverageMonitor via the new class method.
2. Tell the monitor which classes and methods to watch. There are three methods that one can use:
   - cover: aClass—covers all methods for that class (or metaclass)
   - cover: aClass including: aCollection—covers the indicated methods for that class
   - cover: aClass excluding: aCollection—covers all methods for the class except the specified ones
3. Start monitoring these methods, by sending enableCoverage to the monitor.
4. Perform the tests to be monitored.
5. Stop monitoring by sending disableCoverage to the monitor.
6. Inspect the results, using these methods:
   - coveredMethods—returns a list of all methods that were executed during the tests, along with how many times they were sent
   - notCoveredMethods—returns a list of those methods on the covered classes that were not executed
   - browseNotCoveredMethods—brings up a MethodBrowser on those methods that were missed during the test

ARBOR INTELLIGENT SYSTEMS, INC.



VisualWorks makes you productive.

Arbor Help System • Arbor Utilities • Arbor Inspector

make you even more productive!

At Arbor, we've been building Smalltalk applications for over five years.
During that time we've learned quite a bit about what developers need to be productive.
Now we've taken some of that knowledge and packaged it for your team.

Arbor Utilities—Over 50 enhancements and additions to the VisualWorks environment.

Arbor Inspector—An enhanced version of the standard VisualWorks inspector that eliminates the need to open multiple windows while inspecting—no more clutter, no more fuss.

Arbor Help System 3.0—The best just got better. . . For over two years AHS has been the easiest, most powerful way to add end-user help to your application: Context sensitive, widget based help that doesn't need a developer to author • A powerful, hyperlinked on-line documentation browser • Support for multiple languages and easy integration into object databases. . . it's all still there. With version 3.0, we've added numerous features and enhancements to make AHS more 'helpful' and easier to use for developers, authors and end-users alike. Also available for Argos.

Find out why so many companies are turning to Arbor for help.

Call today, be **more productive** tomorrow. Site licensing is available.

(313) 996-4238 • *fax* (313)996-4241 • info@aisys.com

The following code fragment illustrates the use of these methods:

```
| mon today |
mon := SRCoverageMonitor new.
mon cover: Date.
mon enableCoverage.

"This is the test that coverage for which is being measured"
today := Date current.
Transcript show: 'The date is: ', today printString; cr.

mon disableCoverage.
mon browseNotCalledMethods.
```

A few caveats about using these classes:

- Because method wrapping does change classes at a low level, it is best to save your image before using it, in case of disaster. If the test crashes and fails to run to completion, one can restore the original versions of the instrumented methods via SRInstrumentedMethod class>>restoreAllOldMethods. Each wrapped method is tracked in a class variable, which this method uses to unwrap them all.
- Enabling coverage on methods that the coverage analysis classes use will cause an infinite loop. Only enable coverage on application-specific classes or method extensions.
- The enableCoverage and disableCoverage methods may take a while to run if a lot of methods are being monitored.
- Methods whose selectors are made up solely of punctuation (e.g., <=) cannot be wrapped. The algorithm for coming up with a temporary selector (the original with a "real_" prefix) will not work. SRInstrumentedMethod could be modified to use an alternate scheme that would overcome this limitation.

OTHER SMALLTALKS, OTHER COVERAGE METRICS

Coverage analysis is not limited to VST. Both VisualWorks and IBM Smalltalk/VisualAge can employ method coverage using the same technique. Note that configuration management tools, such as ENVY, will require slightly different code, as they typically have different methods for compiling methods into classes. Also, we do not want the instrumented version of the method to go into the revision history, lest it grow out of control.

You could also implement other coverage metrics, but with some difficulty. Ideally, one could do loop and branch coverage by creating modified versions of Boolean and the block classes (e.g., VST's ZeroArgumentBlock). However, most Smalltalks inline a lot of that code, so real messages are not sent; hence, the modified versions would not get triggered. One could modify the Smalltalk compiler classes to overcome this problem.

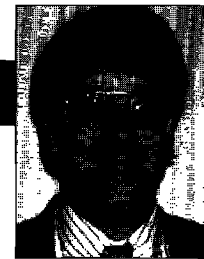
SUMMARY

Coverage analysis is an important component in the quality-assurance program for software development. It is the only real way to feel confident that the test suite is doing its job. These classes for VST will give you a head start toward incorporating method coverage on your project.

Acknowledgments

It should be noted that the research for this article was conducted as part of work done for American Management Systems. Also, the author thanks Doug Kittelsen for his able editorial assistance.

Mark Murphy is an independent consultant and Smalltalk toolsmith. He is also the author of QUALITY TOOLS FOR C/C++ (Prentice Hall, 1995). He can be reached at 71202.2241@compuserve.com.



Bob Hinkle Ralph E. Johnson

Breakpoints revisited*

In the last issue, we described the implementation of a new subsystem for compiling based on the new classes MethodProducer and ParameterizedCompiler. In this issue we describe an extension of this subsystem and use it to implement a more powerful variety of breakpoints. This will provide better motivation for the large degree of flexibility built into ParameterizedCompiler.

Our new breakpoints must be locatable between any two statements. Each breakpoint also has a condition and is activated only if this condition evaluates to true. These new breakpoints have a three-phase lifecycle. First, a breakpoint is created by a user interacting with a programming tool to insert a breakpoint into the text of a method. Second, the breakpoint is implemented when the source text is compiled into a new method. Third, the breakpoint is examined when the new method is browsed, and it's activated when the new method is executed, drawing attention to itself in some useful way. We will focus on the second phase of this lifecycle because that's where the bulk of the work is done and where we use the changes to the MethodProducer-ParameterizedCompiler team.

THE EVOLUTION OF METHODPRODUCER

The design we described last issue used the new classes MethodProducer and ParameterizedCompiler to restructure compilation, making it easier for programmers to specialize the process. Figure 1 shows how these classes interact.

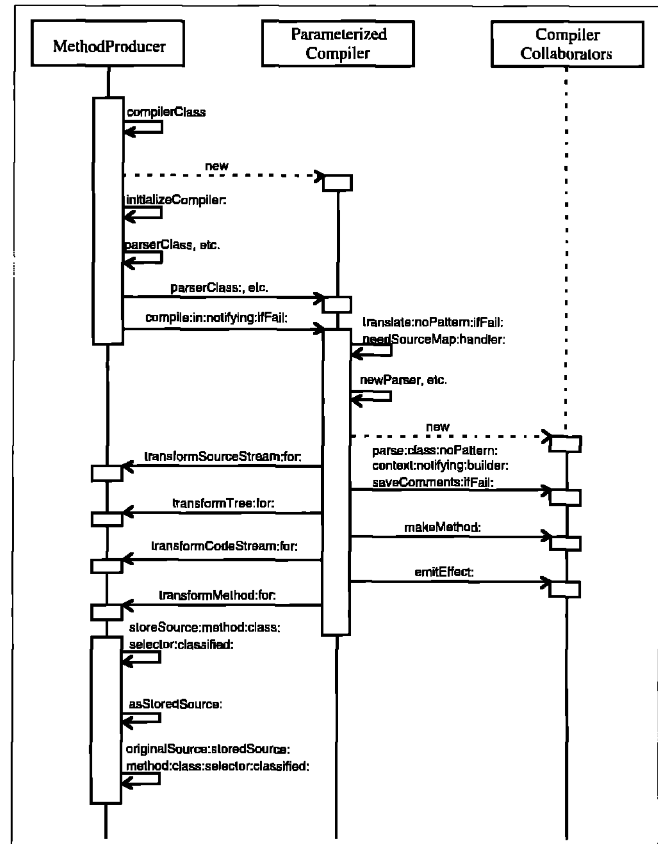


Figure 1. Interaction diagram for MethodProducer and ParameterizedCompiler

This diagram indicates the sequence of messages and activity during method production. Time flows from top to bottom, and vertical boxes indicate periods of activity for a given object or logical group of objects. Solid vertical lines show the lifetime of an object. If an object does not exist at the beginning of the interaction, its vertical line will be dashed until the moment it's instantiated. Solid horizontal lines represent message sends, with the selector indicated, while dashed horizontal lines represent messages that create a new object in the interaction.

This process begins when a MethodProducer creates a new ParameterizedCompiler, passing as parameters the various classes used to create collaborators during compilation. The MethodProducer then has the ParameterizedCompiler begin compiling. The ParameterizedCompiler does this in much the same way the original SmalltalkCompiler behaved, with two exceptions. First, it creates new collaborators using the classes it was given by its MethodProducer. Second, at four points during compilation, the Parameterized-Compiler calls back to its MethodProducer, allowing the MethodProducer to transform the objects flowing along the

Bob Hinkle is an independent Smalltalk consultant and writer. His current focus is the improvement of existing tools and the creation of new tools to revitalize the Smalltalk environment. He can be reached at hinkle@primenet.com. Ralph Johnson learned Smalltalk from the Blue Book in 1984. He wrote his first Smalltalk program in the fall of 1985 when he taught his first course on object-oriented programming and design. He has been a fan of Smalltalk ever since. He is the only author of Design Patterns: Elements of Reusable Object-Oriented Software to regularly program in Smalltalk, and continues to teach courses on object-oriented programming and design at the University of Illinois.

* Source code for the breakpoints package is available by anonymous ftp from st.cs.uiuc.edu. Look for the file **Breakpoint20.st** in **pub/st_vw**. A new version of lightweight classes compatible with this project is also available in the same directory in the file **Lightweight20.st**.

Help Designer

for VisualWorks™

Help Designer is not just a programmer's tool - now any team member can create high quality on-line help. This powerful development tool is rich in features, provides flexible set of tools, and facilitates the reuse of components within your applications. Here is what you get:

Tools

- Help Editor
- Help Viewer
- Image Editor
- Text Editor
- Help Manager
- Control Panel
- Help Custom Controls

Features

- Context-sensitive help
- Inline and outline
- Tag Help
- Hypertext links and references
- Popup definitions
- Keyword search
- History support
- Macro definitions
- Access to font, paragraph, and color attributes
- Embedded objects
- Run-time editing mode
- Platform independent help files

FREE DEMO AVAILABLE !

TO ORDER CALL 212-765-6982

FAX REQUEST 212-765-6920

GreenPoint, Inc.

77 West 55 Street, Suite 110
New York, NY 10019
EMail:75070.3353@compuserve.com

VisualWorks™ is a trademark of ParcPlace Systems

compilation pipeline. When the ParameterizedCompiler returns a new compiled method, the MethodProducer calls back to the target class to load the new method into its method dictionary and to store the method's source code, in whatever way the class deems fit.

The first extension of the subsystem is a new subclass of ParameterizedCompiler called GenericCompiler. Whenever GenericCompiler creates a new collaborator, it sends a message to its producer with the new collaborator and itself as parameters. This allows the producer to initialize the collaborator as desired.

The second extension redesigns MethodProducer to make it less monolithic. Suppose we're designing a MethodProducer for breakpoints, and later we wish to add producers that support activating instance variables and test coverage instrumentation. How would these three producers be combined? We could make three sets of changes to the original MethodProducer, but, as the number of producer specializations grow, this will result in one horribly complicated class. One can imagine methods that have lots of internal tests: if there are breakpoints but no active variables or test coverage, do this; but if there are breakpoints and active variables but no test coverage, do that; and so on. This becomes a programmer's (and especially an object-oriented progammer's) nightmare. A better solution is to create a new hierarchy of method producers (see Fig. 2) that can be combined dynamically at runtime.

Dashed ellipses indicate *abstract* classes, classes that define an interface but are not intended to be directly instantiated. Solid ellipses indicate *concrete* classes, which are to be instantiated. The arrows indicate inheritance. LightweightProducer is a producer associated with lightweight classes, which we described in a previous article.[1]

MethodProducer is now an abstract class defining the interface for all producers. MethodProducer has two subclasses. ClassBasedProducer is the normal default method producer for a class (much like MethodProducer used to be). ProducerModifier is an abstract class. It is a building block for creating new subclasses that can be combined with a ClassBasedProducer to create a composite producer object with new specialized capabilities. Using this architecture, a method producer may be either a single ClassBasedProducer object or a chain of ProducerModifiers connected by their component instance variables and terminating in a ClassBasedProducer. Another ProducerModifier can be added to an existing producer by sending the ProducerModifier the #component: message, which stores the producer in the ProducerModifier's component instance variable, effectively placing the ProducerModifier at the top of the producer chain. The compiler interacts only with the topmost producer, or simply "top producer," thus keeping the nature and organization of the producer structure invisible to the compiler. This way of combining producers may seem familiar, as it's similar to the combination of Wrappers and Views. Both ProducerModifiers and Wrappers are applications of the Decorator pattern described by Gamma et al.[2]

The methods defined in MethodProducer fall into two categories. The first category of methods are called "driver" methods. These are methods that drive the compilation process and should be executed only once, by the top producer. Examples of driver methods are the public interface methods such as #compile:in:notifying: and #parse:in:notifying:, and instance creation methods such as #newCompiler and #newParser. The second category of methods are the "chain" methods. These methods are ones that must be passed down the chain of producers so that each one can provide its own special processing as desired. For

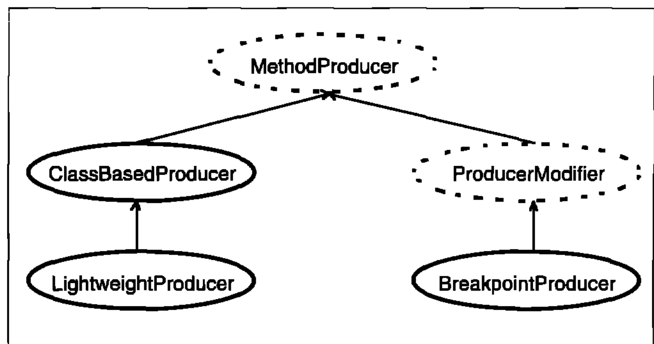


Figure 2. The new MethodProducer hierarchy.

example, the four transformation methods are chain methods. Each ProducerModifier can transform the input, but it must then pass the input on to its component for more possible transformations. The new GenericCompiler sends messages to the top producer to initialize each collaborator, and these messages must also flow down the producer chain.

A third example of chain messages requires a little more thought to implement. These are the various messages, such as #parserClass and #builderClass, used by MethodProducer to specify the classes used by ParameterizedCompiler. In a producer chain, each ProducerModifier and the ClassBasedProducer may have preferences for a given collaborator, and somehow these different preferences must be combined into a single answer to pass to ParameterizedCompiler. We combine classes using the message #composeWith:, which must be implemented in any classes that instantiate compilation collaborators. When this message is sent, the answer must be a class that has the abilities of both the receiver and the message parameter. By default, ClassA composeWith: nil returns ClassA. Furthermore, if ClassB inherits from ClassA, the answer to ClassA composeWith: ClassB should be ClassB. However, if ClassA and ClassB are not related by inheritance, then there must be some class ClassC (and it will probably be a subclass of one of the two) that melds their behavior for compilation, and ClassC should be returned from ClassA composeWith: ClassB. Thus, the result of #composeWith: will be the union of the two classes considered as implementations.

With this architecture, the interaction between a GenericCompiler and a producer chain (see Fig. 3) is quite similar to the interaction of Figure 1. Each ProducerModifier responds to chain messages by performing any special processing of its own and then forwarding to its component. The GenericCompiler, in addition, adds its call-backs to the producer for initializations.

In this diagram, the thick arrows represent interactions between the top-most ProducerModifier and its components. In these cases, the ProducerModifiers sandwich any special-processing code of their own around forwarding the same message to their component. This forwarding process stops at the ClassBasedProducer because it has no components.

MethodProducers have a number of responsibilities to their associated compilers, with default responses defined in MethodProducer and ProducerModifier. As a result, most new specializations of either ClassBasedProducer or ProducerModifier need implement only a few messages. When we were first developing this new design, we worked with a chart that summarized the necessary changes for various kinds of producers. We've reproduced a portion in Table 1. It may give you an idea of how to begin if you'd like to develop a new producer of your own.

The left-hand column contains messages sent to producers from themselves and their associated compiler. The next three columns contain notes about the implementations of these messages for three different kinds of

Deployable Smalltalk Expertise

IBM VisualAge Solutions Provider

Here's Your Chance To Discover What A Smalltalk Consulting Firm Can Really Do.

Object*Intelligence*™

Helping Clients Build Enterprise Applications

- ParcPlace VisualWorks™
- IBM VisualAge™
- Digitalk Visual Smalltalk™

Consulting & Development Services

- Hourly Smalltalk Contracting
- On-Site Smalltalk Development & Project Management
- OODBMS Development: Gemstone™, Versant™ & ObjectStore™
- On-Site Mentoring & Training
- Object Modeling, Analysis & Design

Call 800.789.6595 or

e-mail: info@objectint.com

Object*Intelligence*

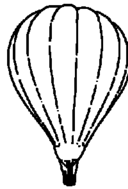
900 Ridgefield Drive, Suite 240
Raleigh, NC 27609
Voice 919.878.6690 Fax 919.878.6695

The Smalltalk Store

405 El Camino Real, #106
Menlo Park, CA 94025, U.S.A.
voice: 1-415-854-5535
or 1-800-ST-SOFTWARE
fax: 1-415-854-2557
BBS: 1-415-854-5581
email: info@smalltalk.com
compuserve: 75046,3160

The Smalltalk Store carries over 75 Smalltalk-related items: compilers, class libraries, books, and development tools. Give us a call or send us an email - we'll put you on the mailing list and send you a copy of our combination newsletter-catalog. It's informative and entertaining.

When you get the chance, check out our new dialect-neutral Smalltalk bulletin board system at 415-854-5581, 8N1.



Send For Our Free Catalog!

producers. Cells left blank indicate that the default response (as implemented in MethodProducer) is used. An equals sign "=" means the implementation is the same as in the cell immediately to the left.

BREAKPOINT COMPILATION

With these new additions to the producer-compiler family, we can now compile our improved breakpoints, using the class BreakpointProducer. BreakpointProducer is a subclass of ProducerModifier because it will be needed if and only if there are breakpoints present in the text being compiled. As we said in the previous section, a new ProducerModifier only needs to override those few methods where it must intervene to fulfill its purpose. In the case of BreakpointProducer, the necessary methods can be determined by a quick glance at its column in Table 1.

It has four preferences for collaborator classes, using the new classes GenericCompiler, GenericCodeStream, ExtendedNodeBuilder and AlteredDisplayMethod. GenericCompiler is necessary because Table 1 indicates that BreakpointProducer must perform some collaborator initializations. GenericCodeStream is a partner of GenericCompiler that makes it possible to initialize CompiledMethods and CompiledBlocks as they're created. BreakpointProducer uses ExtendedNodeBuilder to create a new kind of parse node called BreakpointNode, whose purpose will be explained below. AlteredDisplayMethod is a new subclass of method that maintains

Table 1. Implementation notes for different producers.

	ClassBasedProducer	LightweightProducer	BreakpointProducer
compilerClass	ParameterizedCompiler	=	GenericCompiler
initializeCompiler:			
parserClass	Parser	=	
initializeParser:for:			ignore breakpoint characters
builderClass	ProgramNodeBuilder	=	ExtendedNodeBuilder
initializeBuilder:for:			
codeStreamClass	CodeStream	=	GenericCodeStream
initializeCodeStream:for:			
nameScopeClass	NameScope	=	
initializeNameScope:for:			
methodClass	CompiledMethod	UnstoredMethod	AlteredDisplayMethod
initializeMethod:for:			set method's breakpoint flag to true
blockClass	CompiledBlock	=	
initializeBlock:for:			
transformSourceStream:for:			re-position breakpoints to statement end
transformTree:for:			insert breakpoint code into parse tree
transformCodeStream:for:			
transformMethod:for:			
asStoredSource:			strip breakpoint characters from stored version of source
originalSource:storedSource: method:class:selector: classified:		set method's source to equal the original source	set method's display source to equal the original source

two versions of its source, one as stored on disk and the other as displayed in the browser. This allows methods created by the BreakpointProducer to show breakpoints on the screen without saving them to the Change File.

BreakpointProducer also implements two initialization messages. The first used during compilation is:

```
initializeParser: aParser for: aCompiler
    ^(component initializeParser: aParser for: aCompiler)
        typeTableAt:
            Breakpoint absoluteBreakpointCharacter
        put: #xIgnore;
        typeTableAt:
            Breakpoint conditionalBreakpointCharacter
        put: #xIgnore;
        yourself
```

This code causes the Parser to ignore the special breakpoint characters, which permits them to occur anywhere within their statement, including in the midst of other tokens, without causing an error. Note that this method forwards the initialization method to BreakpointProducer's component. This forwarding must be done in every reimplementation of a chain method to ensure correctness. The second initialization used by BreakpointProducer is:

```
initializeMethod: aMethod for: aCompiler
    ^(component initializeMethod: aMethod for:
aCompiler)
        breakpoint: true;
        yourself
```

This sets a flag associated with InstrumentedMethod, AlteredDisplayMethod's superclass, to indicate that the created method has breakpoints. The Browser uses this flag to indicate breakpointed methods in its selector list. It could also be used by new tools to track debugging changes in the environment and to turn breakpoints on or off on a class or project basis.

Two more methods are required to handle the source code as it's being stored. First, to strip breakpoint characters from the stored version of the code, BreakpointProducer implements:

```
asStoredSource: code
    ^(component asStoredSource: code) copyWithoutAll:
        (Array
            with: Breakpoint absoluteBreakpointCharacter
            with: Breakpoint
conditionalBreakpointCharacter)
```

Then, before the class is sent the message to store source, each producer in the producer chain receives the message #originalSource:storedSource:method:class:selector:classified:. This message gives producers a chance to do special processing. BreakpointProducer sends the new method the #displaySource: message with the original source (that is, the source as actually parsed, before the breakpoint characters were stripped out) as parameter. That way, the new

Get CORBA 2.0 Interoperability *Now* with HP DST

Need to create 3-tier, enterprise-wide applications and integrate other languages with your Smalltalk application?

With HP distributed Smalltalk 5.0, you can move beyond simple client/server to true distributed, enterprise-wide applications. That's because you get tools for distributed development and debugging, a CORBA 2.0 object request broker, and related object services that make it easy to create business objects and distribute them wherever you like on your network. Control your business objects with the Transaction CORBAservice in HP DST. Integrate them with other C++ objects when you use HP DST and another CORBA 2.0 object request broker.

HP Distributed Smalltalk is an extension of the ParcPlace VisualWorks environment. Put together, your programming team gets a faster, easier way to develop and deploy distributed applications on any combination of supported UNIX and PC platforms.

Send us your name, address, and phone # and we'll send you free white papers titled "Manager's Guide to Distributed Objects" and "HP DST Technical Information."

Phone: (408) 447-4722
FAX: (970) 229-2180
Attention: HP DST White Papers
e-mail: dst@sde.hp.com

© 1995 Hewlett-Packard Company

method can provide two forms of its source text, both with and without breakpoints included.

BreakpointProducer's most complicated responsibility is to respond to the #transformTree:for: message. It must transform the input parse tree by adding new code at every spot where a breakpoint occurs. We implemented BreakpointProducer's response using three methods. The implementation starts with:

```
transformTree: methodNode for: aCompiler
    | block statements |
    source computeTextPositions.
    (self breakpoints asSortedCollection: [ :x :y | x start >-
y start]) Do: [ :bp |
        block := self findBlockEnclosing: bp in:
methodNode.
        block body isSequenceNode
            ifFalse: [
                block
                    arguments: block arguments
                    body: (aCompiler translateBuilder
                        newSequenceStatements:
(OrderedCollection with: block body))].
        statements := block body statements.
        (self add: bp toStatements: statements for:
    aCompiler) isNil
            ifTrue: [^nil]].
    ^methodNode
```

For each breakpoint, the BreakpointProducer sends itself the message #findBlockEnclosing:in: to find the smallest (or most deeply nested) block enclosing the breakpoint. This is done using the ProgramNodeEvaluator we described last issue. The BreakpointProducer then ensures that this enclosing block has a SequenceNode (rather than some individual statement node) for its body. This is necessary because the breakpoint code will be added to the block's body as a separate statement. After that, the producer sends itself #add:toStatements:in:, which we'll explain a piece at a time.

The method first determines which individual statement contains the breakpoint to be added.

```
add: bp toStatements: statements for: aCompiler
    | last c stmt stmts |
    last := (c := statements select: [ :s | s extraPosition
first < bp start]) isEmpty
            ifTrue: [nil]
            ifFalse: [c last].
```

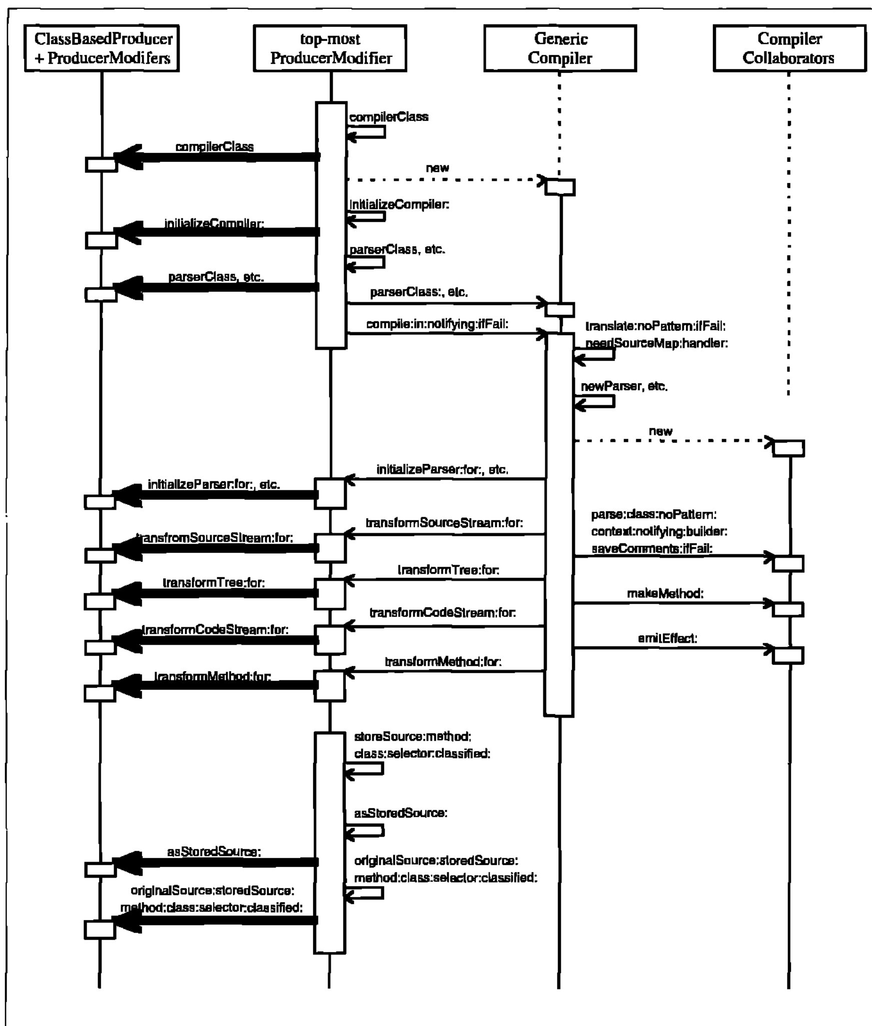


Figure 3. Interaction diagram for ProducerModifiers and GenericCompiler.

We added a new positional instance variable extraPosition to all ProgramNodes to help us find the statement containing a given breakpoint. (The existing variable sourcePosition is unhelpful because it doesn't run from the statement's beginning to end, but just over those characters hightlighted during debugger stepping.) This variable is set during the parsing of a SequenceNode, and it includes all characters in each statement, from the first non-whitespace character to the last one, including the terminating period if present. With this new information, we might wish simply to test which statement's extraPosition contains the breakpoint's position. Unfortunately, extraPosition does not include white spaces between statements (and furthermore it would be quite difficult to make it do so). So, instead, we find all statements that start before the breakpoint is defined, and choose the last of these.

```
stmts := aCompiler translateParser
    parseBody: bp sourceString readStream
    class: aCompiler translateClass
    notifying: SilentCompilerErrorHandler new
    ifFail: [^nil].
```

Next we parse the breakpoint's source

string, using the same parser as for the code that contains this breakpoint. The breakpoint's response to #sourceString will either be 'self halt' or '[some condition] value ifTrue: [self halt]', where 'some condition' is the condition string input for a conditional breakpoint. (Using #halt to activate breakpoints is a matter of convenience. You could instead implement a new Signal for breakpoint activation, which would allow special handling and interface for breakpoints.) If there is a problem parsing this conditional part of the code, the method returns nil to MethodProducer>>transformTree:for:, which can indicate an appropriate error condition. We assume most condition code will be relatively simple, and therefore that this situation will not arise frequently. If it does, you can add more elaborate error handling, using a different requestor to the parsing, to be able to report more fully whatever error occurs.

```
stmt := stmts at: 1.
(stmt nodeEvaluatorClass new tree: stmt)
   do: [ :n |
      n sourcePosition: (bp start to: bp stop);
         extraPosition: (bp start to: bp stop)].
bp isConditional
   ifTrue: [stmt receiver sourcePosition: nil].
```

This step adjusts the sourcePosition and extraPosition of the ProgramNodes generated from the breakpoint's source string so that they point only to the breakpoint's character position in the method's source string. Otherwise, these nodes would refer to indexes from the breakpoint's source string that don't make sense in the method's string. Also, we set the sourcePosition of a conditional breakpoint's condition block to nil, so that it won't be present in the sourceMap used for debugging. As a result, every breakpoint will require only a single step to process in the debugger, making the interface uniform.

```
stmt := aCompiler translateBuilder newBreakpointBody: stmt.
stmt extraPosition: (bp start to: bp stop).
```

The above two lines wrap the subtree parsed from the breakpoint's source string inside an instance of BreakpointNode, the new kind of ProgramNode mentioned above. BreakpointNode is a subclass of InstrumentationNode, which is in turn a subclass of InvisibleNode. The latter class is an important addition to the ProgramNode hierarchy, as it creates a class of statements that can be added to a parse tree with no visible effect. Without something like InvisibleNode, a reflective programmer would have to be very careful where they added instrumentation statements into a parse tree, to respect the semantics of block return. These semantics state that the return value of a block is the value returned by the last statement in the block; or, in the case of a block with no statements, the value of the last block parameter; or nil in the case of a block with neither statements nor parameters. We changed the two places that implement these semantics to use the notion of "visible statement(s)" in place of "statement(s)", thus allowing us to insert new invisible statements anywhere without affecting the code's visible behavior.

```
last isNil
   ifTrue: [statements addFirst: stmt]
   ifFalse: [statements add: stmt after: last]
```

The final step in #add:toStatements:in: adds the newly produced BreakpointNode into the collection of statements that was passed in. It's added after the statement that contains the breakpoint, if there is one, or else before all statements. Breakpoint code might be added after a return with this mechanism, since that error is only detected during parsing. This will not cause any errors, although of course the breakpoint will never be activated. It would be possible to alert users if this happens, so they won't be surprised by breakpoints that don't activate when it seems they should.

CONCLUSION

These new breakpoints improve significantly on the old version, and, when combined with the lightweight classes of our previous article, further extend the programmer's options and opportunities when debugging. These improvements can be characterized as locality enhancements. With breakpoints that can be conditionally

continued on page 18

Database Solution for Smalltalk

A class library for ODBC Database Access

- ODBC 2.x support for 50+databases
- Visual development components for database access
- Native ODBC data type support
- Online documentation, source included, no runtime fees
- programming examples and sample application
- **OO to RDBMS** mapping framework, based on types & brokers, ideal for complex client-server applications
- compatible with OTI's ENVY/Developer, Object Share's WindowBuilderPro
- SLL and Team/V packaging support

Versions Available for Windows, Windows-NT, OS/2, and for IBM, Digitalk, and ParcPlace

New for ParcPlace VisualWorks

Tel: 416-787-5290
Fax: 416-797-9214
CompuServe: 73055,123
Internet:73055,123@compuserve.com

Check out LPC's Internet World Wide Web home page: http://www.rwi.com/smalltalk/products/vendors/lpc/lpchome.html

Jan Steinman

Barbara Yates

Exploiting stability

As Dave Thomas of Object Technology International is fond of saying, "Software development is a 'bursty' process." Long periods of time may pass during which seemingly nothing is accomplished, followed by periods of intense development and miraculous productivity.

When one person works alone, this "burstiness" averages out, and its impact is limited to "did I get done what I wanted to in the time I had?" However, when the efforts of many need to be coordinated, the "bursty" nature of development can sink a project, or at least significantly impact its schedule.

ADOPT A SPIRAL PROCESS

Traditional software development follows a "waterfall" process, in which different kinds of activities (such as specification, design, implementation, and test) are allotted sequential time periods, with rigidly defined checkpoints at the end of each phase.

Waterfall works great if you can determine exactly what you want to build, but most modern systems are not so easily specified, or their requirements change during development.

Barry Boehm recognized this weakness and proposed a scheme by which the various development activities are more tightly integrated. Such a "spiral" process can be classified as *iterative*, in which portions of a system are reimplemented in each cycle to achieve quality goals, or *incremental*, in which a system is grown by adding functionality in each cycle. Ideally, these two patterns should be mixed and used as needed.

This spiral process has numerous advantages, mostly in providing the flexibility needed to apply Smalltalk to ill-defined or evolving systems, but there is no free lunch. A spiral process falls down when it is too rigid, or when it must closely follow a corporate waterfall process mandate.

Waterfall is falling out of vogue (overheard at OOPSLA: "If you're not doing *incremental* development, you're doing *excremental* development..."), but for well-defined problems with extremely high reliability requirements, waterfall may be more appropriate. If your software is life-critical, you'd better fully understand your requirements up front, and the flexibility of spiral development is then not as useful.

AVOID SCHEDULE RIGIDITY

In waterfall development, the predefined project phases naturally break a project into manageable bits. In spiral development, there is no such natural division, and an arbitrary division is often used. These "cycles" are typically assigned the same length, such as seven weeks.

The functionality assigned to a cycle will be based, to some extent, on guesswork, especially in an organization's first spiral process project. This shouldn't reflect poorly on the manager; rather it is an acknowledgment of what *really* happens in most waterfall-based projects! It should help to know ahead of time that the spiral process supports this inherent "truth in planning," and that functionality *will* be shuffled around to different cycles as its meaning and relationships are elaborated.

However, you don't get the full benefit of spiral's flexibility if you are rigid about cycle length. If your project intends to exploit stable periods, some cycles may take less than their allotted time, while few (or none!) should take extra time.

When deciding when to end a cycle, it's better to postpone functionality than extend the cycle. If a particular feature has not even been started before the last week of a cycle, slip it; don't cram it in!

If you are rigid about *both* functionality and cycle length, you are simply going to "fail" most of the time—you deserve a more realistic definition of success when working on ill-defined problems!

If you are rigid about functionality, you may end up in the "death spiral," where each extension of the cycle lets you discover problems in the specification or implementation of the functionality, so that you keep trying to squeeze in more work.

The quality of a rigid-functionality cycle's product suffers because the lengthened cycle has shifted emphasis from integrated development to simple coding, at a time when the emphasis *should* be on integration, testing, and

Jan Steinman and Barbara Yates are cofounders of Bytesmiths, a technical services company that has been helping companies adopt Smalltalk since 1987. Between them, they have over 20 years Smalltalk experience. They can be reached at Barbara.Bytesmiths@acm.org or Jan.Bytesmiths@acm.org.

documentation clean-up. The result is a mad push to add a lot of code that will most likely have to be rewritten in the next cycle anyway!

Managers who are new to the spiral approach are often reluctant to shorten a development cycle and postpone functionality, due to "waterfall-think," e.g., "if it doesn't get in now, it'll be dropped altogether and we *must have* the 'DithyWither' function!" Spiral's flexibility means that the relative importance of different features can be continuously reevaluated, so that if it is really that important, "DithyWither" can be the first thing to happen in the next cycle.

WHAT IS STABILITY?

The definition of stability is like a Supreme Court Justice's definition of pornography: "I can't tell you what it is, but I know it when I see it!" That isn't good enough, so here's a working definition: Stability is a condition where features are implemented, integrated, and tested to a planned degree of completion, combined with low rates of recent change.

The combination of a *planned degree of completion* and *lack of change* is crucial; things that are completed—including testing—but have recently changed considerably should not be considered stable. Conversely, incomplete things that have not changed in a long time are not really stable, due to incipient change.

Stability is a condition where features are implemented, integrated, and tested to a planned degree of completion, combined with low rates of recent change.

Also note that something does not have to be in its end form to be stable—otherwise, the only stable point to exploit would be at project end! Alternating incremental cycles with iterative cycles can provide stable points even during evolution.

Features at a planned degree of completion...

Here's an example of planned completion: John's stuff works with Sue's stuff and both of theirs works with Ed's stuff. Each of them has completed most—and perhaps all—of their assigned features for the cycle.

If *all* of them have completed less than 3/4 of their tasks and it's one week before cycle end, the cycle plan was far too ambitious. If *any* of them are in this situation, it's time to slip some of their work into the next cycle.

Sometimes, the last week of the cycle arrives, and individual modules are complete but not yet integrated with the entire team's development. If this begins to happen regularly, it is a sign that your project is not practicing *continuous integration*, but is instead doing "mini-waterfall" development. It may be *implemented* and *tested*, but without being *integrated* such work is still not stable, and your developers need to review your groupware support to see if they can do internal integration more frequently.

Low rates of recent change...

If your team is honest and communicating, it is not hard to assess the feature-completion aspect of stability, but you'll need some tool support to accurately assess change rate, sometimes derisively called "code thrash."

With groupware tools such as Team/V or ENVY/-Developer, it is not too difficult to measure change rate. Even ad hoc schemes using file-based repositories like SCCS can give you an idea of your change rate, albeit at a gross level.

Using ENVY/Developer information, we measure the ratio of method editions to methods (me/m) between any two app or subapp editions. Obviously, no changes would yield an me/m of one, but we've seen numbers as high as 20 for highly changing code! Although we've made no formal study, a year or so of gathering this metric leads us to believe that "stable" code will have an me/m below 2.

Like all "sadistics," this one is subject to artifacts and abuse. New Smalltalkers and those unfamiliar with ENVY tend to have inflated me/m rates not necessarily because their code is unstable but simply because they insert **halt** more often, or because they manually revert code back to the way it was rather than loading a previous edition—they are merely less efficient users of the environment.

Conversely, paranoid developers may go out of their way to avoid seeming "unstable" by doing lots of work in workspaces, working on multiple methods at the same time, or not saving methods as often as needed. Avoid this by building a group culture where measurements are indicators, not juries!

Rather than reducing an entire cycle to a scalar ratio, it would be interesting to "bucketize" method timestamps in a module and plot the resulting histogram against time. A bucket size of one day should be adequate, and will make tallying method timestamps easier. This data gives a manager an indication of intracycle stable points rather than a gross "how stable are we right now?" figure.

I'M STABLE, WHAT NOW?

With modern code-management systems, there is no reason for your entire team to "freeze" their development when they near the end of a cycle. Exploiting stability necessarily means dealing with asynchrony within the team.

If John overestimated his work, and so his stuff is deemed stable two weeks before the planned cycle end, what should he do now?

(a) Read news or play Doom.
(b) Mess around with his code, making it run faster or take less space.
(c) Continue with work allotted to the next cycle.

What John *wants* to do is (a), but that usually doesn't work out! What he *should* do is (c), but people are usually afraid

of destabilizing things by adding new features. What *usually happens* is (b), which often destabilizes things as much as (c) would have!

The key is for John to "checkpoint" his stable work before getting a jump on the next cycle's work. In ENVY, this means making certain that all his stable code is released, but *none* of his new code gets released. In extreme cases, this may require working in a new app or subapp edition (or even a different image), although it is best to avoid having more than one editable edition at any given time.

Periods of stability are also ideal points for peer review, although not necessarily on the stable modules! Achieving stability gives developers a chance to plan for their next period of instability, which takes some of the uncertainty away. *Design* review should be conducted on planned work, and *code* review should be conducted on stable or nearly stable work.

Unfortunately, peer review is often done the other way around! There is a strong tendency to review designs once the code is written and stable, when no one is willing to destabilize them, and an equally strong tendency to review code that is still in flux, which means very little of the reviewed code will actually end up in the product because most of it will change. Ignoring stability concerns when planning peer reviews simply wastes the time of the reviewers and gives a false sense of security.

It is important to be able to recognize incipient instability, or all this flexible planning and asynchronous development goes away. If John suddenly recognizes a design error or discovers a new requirement, he is about to leave stability. If Sue needs to make small changes to a class that handles function "furble," while Ed needs to make a small change to that class to improve behavior for "snotzer," their integration stability decreases, even though their individual changes are benign.

In such cases, it's best to call an early cycle end so that you can capture your stable point before everything comes tumbling down!

PLANNING FOR STABILITY

We hinted earlier that elements of both incremental and iterative spiral methods can be successfully combined. We call this *expand-contract* development, and it is a lifesaver for projects that encounter quality problems related to "rampant featuritis."

In this scheme, periods of added functionality are interspersed with periods whose purpose is to improve the quality of existing functionality. "Quality" is an overloaded term, and any aspect of it could be pursued in intervening periods. Typical quality goals are increased factoring, increased abstraction, increased cohesion, and increased reuse, as well as decreased mass of code, reduced coupling, and reduced variable scope. These periods need not be entire cycles, nor does the entire team need to switch modes in lock step.

Unfortunately, most projects cannot tolerate (or think they cannot tolerate!) budgeting as much as half their time to rework: "You mean that every six weeks, you're going to take six weeks to rewrite what you just did? Why don't you guys just take a 10-minute coffee-break every 10 minutes?!"

A contract phase by itself does not yield stability, because it is itself changing code. Stability is like fine wine or compound interest: it requires time. By shifting the goal from "more" to "better," the output of a contract phase is much more likely to be deemed stable after one additional cycle than the output of an expand phase.

CONCLUSION

A spiral development process can be useful for the flexibility it provides to reevaluate and reschedule work. Typically, this flexibility results in *increased* work in a cycle, but rarely results in *reduced* work in a cycle. This is a missed opportunity!

When a cycle somehow ends up with fewer days than expected, many managers sacrifice quality and stability for functionality, resulting in massive rework in subsequent cycles.

However, with careful attention to points of stability during a cycle, a manager can improve quality, morale, and productivity by shortening cycles as needed, and individuals in the habit of honest self-assessment can help their manager synchronize the group's efforts at the most stable point.

DEEP IN THE HEART OF SMALLTALK

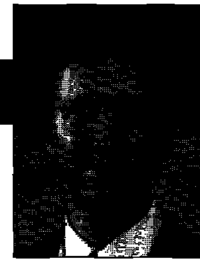
continued from page 15

activated and flexibly located, programmers can investigate close to suspected problems, without having to wade through extraneous debugging information. Likewise, lightweight classes help narrow the focus of debugging to the locality of individual objects.

In addition to improving debugging, this project again demonstrates the large degree of openness in Smalltalk-80, and the usefulness of allowing programmers reflective access to their environment. The central work described in this article is the extension of the MethodProducer-ParameterizedCompiler tandem. We can create them and use them to produce breakpointed methods because all collaborators in the compilation process are first-class objects. The pair of producer and parametrized compiler have many uses beyond breakpoints, and we intend to bring those out in future columns, starting with the implementation of active variables and their use in defining watchpoints in the debugger.

References

1. Hinkle, B., V. Jones, and R.E. Johnson. Debugging objects, THE SMALLTALK REPORT 2(9), 1993.
2. Gamma, E. et al. DESIGN PATTERNS: ELEMENTS OF REUSABLE OBJECT-ORIENTED SOFTWARE, Addison-Wesley, Reading, MA, 1994.



Improving your designs

Mark Lorenz

WE ALL KNOW that successfully moving a project to object technology involves many different efforts and challenges. It is not possible to staff a commercial project of any size with experienced O-O developers. And there are precious few products to help us be successful.

One area that can directly help you develop higher-quality systems using object technology is *metrics*. This article will present a metrics process that you can use to measure how well you and your project team members are using inheritance, collaboration, encapsulation, and O-O design techniques.

A DESIGN IMPROVEMENT PROCESS

The basic steps to a better design are detailed in the following sections.

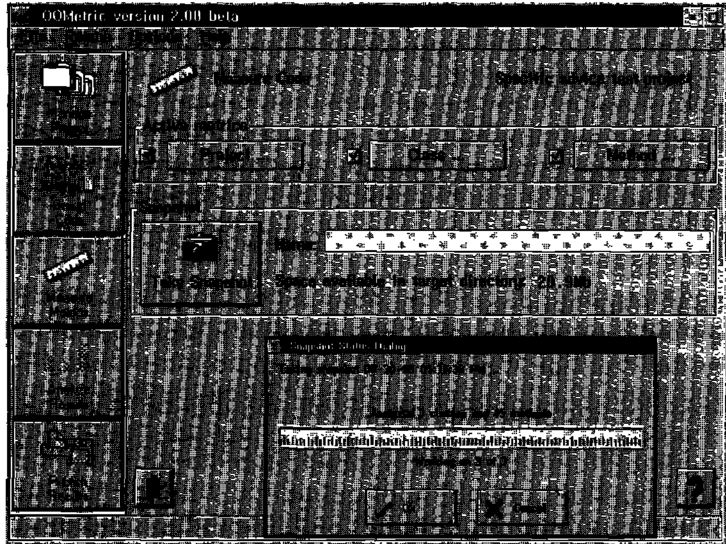


Figure 1. Running a metrics analysis using OOMetric.

Run a metrics analysis

Choose what metrics as well as what target designs to examine and run a metric analysis, as shown in Figure 1.

Interpreting the analysis results:

- Average = the average measurement value for this metric.
- Percent anomalies = the percentage of target objects that are considered anomalies with the current thresholds.
- Maximum value = the largest measurement found in the target design.
- Total = the sum of all measurements taken for this metric.

Look for anomalies

Either online, as shown in Figure 2, or offline, as shown in Figures 3–5, examine the results of your analysis to see what anomalies you find. These are the areas that are outside the thresholds set for your project. In other words, they are possible problem areas that would benefit from design changes.

Figures 3–5 are taken from OOMetric's summary report output, for use with any word processor (in this case, I used Lotus AmiPro). The following subsections take you through some sample analysis results.

Method size: Number of message sends. Figure 3 shows example results. I use this metric instead of others, such as lines of code (LOC), because it automatically removes style issues.

When examining results, don't worry about GUI construction methods or accessing methods (variable getters and setters). Model classes should have smaller methods than GUI classes, which often have a significant number of methods to handle listboxes, buttons...

Smaller methods are more reusable, easier to build and maintain, and indicative of a better O-O design (O-O is about collaboration, even within a class). Reuse is achieved more easily through smaller components, since they do one thing only. Complexity is managed better by delegating to objects that balance the work amongst them.

Method complexity: McCabe complexity. Figure 4 shows example complexity results. Similar to method size, complexity is indicative of methods trying to do too much work themselves rather than delegating to other objects. It is also possibly indicative of poor designs, as evidenced by additional conditional statements instead of the more single-minded logic that occurs with better responsibility distribution.

Mark Lorenz is Founder and President of Hatteras Software Inc., a company that offers education, consulting, and products to help other companies successfully use object technology, as evidenced by commercial products such as IBM's StorePlace and Hatteras' OOMetric. He welcomes questions and comments via email at mark@hatteras.com, phonemail at 919.319.3816, fax at 919.319.3877, or snail mail at 2000 Regency Pkwy., Ste. 230, Cary, NC 27511.

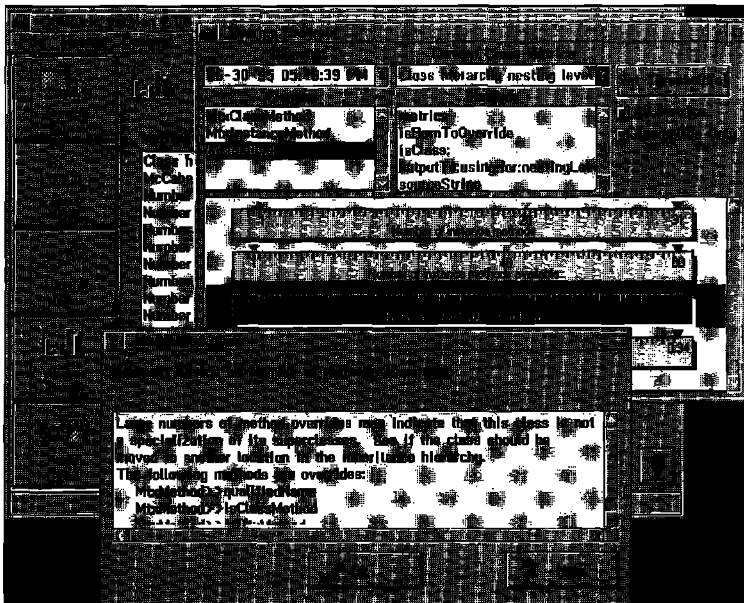


Figure 2. Analyzing metric analysis results for anomalies.

Inheritance: Number of methods overridden. Figure 5 takes a look at the use of inheritance by a project. Framework template methods and method extensions have already been filtered out. What we want to do is also filter out tool-related methods, such as GUI construction methods.

Use anomalies to focus design reviews

Typically, there are sets of classes and methods that raise the anomaly flag over and over again. These are the areas on which you want to focus your attention first and foremost for design reviews. These are the areas that will give you the most "bang for the buck." You can then decide how much time you will spend reviewing other areas, up to reviewing the entire system.

During the review, you need to decide which anomalies are problems and which ones do not need any follow-up actions. Note that anomalies can be due to trends in measurement values and comparisons of measurement values to other projects or company standards.

Design reviews should always result in code volume reduction. It is typical to see methods reduced to 1/3 their original size.

Take actions to improve designs

For those areas that are considered problems, follow-up actions are defined in the design review. Actions taken depend on the types of problems found, of course. At the highest level, however, they fall into two categories: (1) reworking the design and implementation, and (2) dealing with people skills. The latter is not the topic of this article, but generally involves additional mentoring and/or reassignment to non-O-O tasks.

The following sections give examples of actions taken for the anomalies previously discussed.

Message sends. Examining the code for the large methods reveals that they are indeed written more as function-oriented rather than object-oriented logic. In other words, they do a lot of work in a single method instead of leveraging the services of other objects in the business model. They need to be redesigned.

Complexity. Listing 1 gives the source code related to one of the methods that was above the complexity metric upper threshold. A variety of actions to improve the situation are possible, including (1) invocation of methods to handle the outputting of methods and variables:

```
self writeMethodsTo: aStream.
self writeVariablesTo: aStream.
```

These two lines would replace the four loops in the code, while creating additional reusable services at the same time.

Number of message sends

Description: The number of message sends in a method. Message sends are a style-independent way to measure method size.

Average:	4.57
Percent anomalies:	3.05
Maximum value:	135.00
Total:	1948.00

Value	
68.00	BomClass>>reportDatesCreated: method
79.00	BomClass>>writeCpp method
42.00	BomClass>>writeCppHeaderTo: method
57.00	BomClass>>writeSmalltalkClassDeclarationTo: method
29.00	BomClass>>writeSmalltalkMethodsTo: method

Figure 3. Analyzing metric analysis results for anomalies (message sends).

Listing 1. Design review actions (complexity).

```
writeCpp
    "Gen C++ code & write it to a separate file based on
    my name."

  | aFileStream |
  ( self parent isNil ) ifTrue: [ ^self ].     "Object is a
    special case..."
  ( self filename isNil ) ifTrue: [ self filename: ...
  aFileStream := File pathName: self filename.
  ( self allPublicMethods ) do: [ :each | aFileStream
    nextPutAll: ...
  ( self allPrivateMethods ) do: [ :each | aFileStream
    nextPutAll: ...
  ( self allClassVariables ) do: [ :each | aFileStream
    nextPutAll: ...
  ( self allInstanceVariables ) do: [ :each | aFileStream
    nextPutAll: ...
  . . .
```

McCabe cyclomatic complexity (for methods)

Description: Method complexity, as determined by the control flow graph, which reduces to the number of decision points.

Average: 1.58
Maximum value: 13.00
Total: 674.00
Value:

6.00	BomClass>>addContract: method
5.00	BomClass>>delete method
5.00	BomClass>>generateConflictReportFor: method
13.00	BomClass>>reportDatesCreated: method
7.00	BomClass>>writeCpp method

Figure 4. Analyzing metric analysis results for anomalies (complexity).

(2) Validation of the information in separate method, separating out the checks at the beginning:

self **checkValidity.**

This is also a new reusable service.

These actions would bring the complexity level (as well as the method size level) well within the limits set by the project. At the same time, the design is clearer, easier to maintain, and more reusable.

Overrides. As we said, tool-generated methods (e.g., constructWindow for WindowBuilder Pro and abtBuildInternals for VisualAge) should be ignored. In the class from Figure 5, createViews is a GUI construction method. label is supposed to be a framework template, but was written with the wrong Smalltalk selector (subclassResponsibility instead of implementedBySubclass). It was fixed. The other methods need further examination in relation to the class hierarchy and collaborating classes to see if changes are warranted.

Iterate the previous steps

Improving your designs can be a part of each iteration in your development process. Developers can use it during line item production to periodically check their work; team leads can use it during each iteration assessment period to focus design reviews, as we have discussed; and project managers can use it to assess the quality and progress of the project as a whole and compared to other O-O projects in the company. In other words, design quality is an on-going effort and not just an end-of-project report to management.

SUMMARY

We have examined the steps used to achieve a higher-quality OO software system:

- Run a metrics analysis
- Look for anomalies
- Use anomalies to focus design reviews
- Take actions to improve designs
- Iterate the previous steps

Number of methods overridden

Description: The number of methods defined in a class that are also defined in one or more superclass(es). Methods that invoke the superclass' method or override template methods are not included.

Average: 4.31
Percent anomalies: 31.58
Maximum value: 14.00
Total: 56.00
Value

5.00 BomAnalysisModelBrowser class

Specific advice: The following methods are overrides:

BomAnalysisModelBrowser>>initPanes
BomAnalysisModelBrowser>>createViews
BomAnalysisModelBrowser>>activeObject:
BomAnalysisModelBrowser>>applyIndentionString:
BomAnalysisModelBrowser>>label

Figure 5. Analyzing metric analysis results for anomalies (method overrides)

These steps lead to higher-quality designs, which result in systems easier to build and maintain. In other words, they lead to quicker product cycles and more delivered functionality.

Terminology

anomaly A deviation from the common result.

measurement The determination of the value of a particular metric for an object.

method extension A method that invokes the superclass' corresponding method and then adds specialized logic.

metric A standard of measurement used to judge the attributes of something being measured, such as quality or complexity, in an objective manner.

metric analysis Application of a set of metrics to a portion of a software system, resulting in values which can be examined for trends and anomalies.

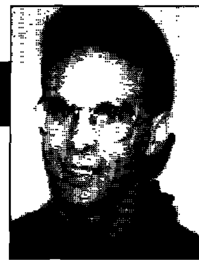
problem A development decision that has been deemed to be low-quality and in need of improvement.

template method A method that designates an empty portion of a design framework that is intended to be filled in by client subclasses.

threshold A measurement value that has been determined through project experiences to be significant in terms of desirable or undesirable designs, with some margin of error.

References

1. Lorenz, M. Object-Oriented Software Metrics, Prentice Hall, Englewood Cliffs, NJ, 1994.
2. Lorenz, M. Rapid Software Development with Smalltalk, SIGS Books, New York, 1995.
3. Hatteras Software. Object-Oriented Metric Workshop course materials, 1995.



Class versioning and instance migration

Jay Almarode

NO MATTER HOW CAREFULLY you design an application, it is inevitable that you will have to change your code in response to new requirements, bugs, or performance bottlenecks.

One of the nice things about Smalltalk is the dynamic nature of application development. You can easily modify a class definition and test the new code immediately. In some cases, you can modify a class while its instances are being used in a running application. This is one reason Smalltalk development is known for its rapid prototyping and high productivity.

All Smalltalk systems allow a developer to add, remove, or modify methods dynamically during development. This is not hard to provide because method lookup occurs at runtime. A more problematic situation arises when a class is modified so that its instances have a different structure. This occurs when a class definition has an instance variable added or removed, or possibly when its position in the class hierarchy is changed (i.e., it is given a different superclass). This column discusses the different strategies used when class modification causes structural changes to instances.

In single-user Smalltalks, the underlying systems only allow one class with the same name to exist. This is appropriate since there is only one name space in which the class could reside. When you redefine an existing class, a new class is created, a "become:" operation is performed to cause the new class to have the identity of the old class, and the new class replaces the old class in the SystemDictionary. But what happens to the instances of the old class? Their state may not match what is expected from them by their own class. Single-user Smalltalk systems handle this situation by automatically migrating all instances of the old class to have the appropriate structure of the new class. Instance variables are removed if necessary, and added instance variables have an initial value of nil.

In multi-user Smalltalk, it is possible for multiple classes with the same name to exist. This is because multiple users have multiple name spaces; so there is no conflict in having two or more classes with the same name. When a developer creates a class with the same name as an existing class, it is important that the old class still exist and its instances behave correctly. This is because other users may be running applications that expect instances of the class to operate a certain way. In single-user Smalltalk systems, modifying a class only affects your work; in multi-user Smalltalk, your changes may affect others. However, developers still need a way to modify a class that is being used by other users without breaking existing applications.

The solution is to create a new version of the class rather than modifying the existing class. The new version has its own unique identity and does not interfere with the operation of the original class. The instances of the original class are unchanged and still look and behave as defined by the original class definition. In GemStone Smalltalk, versions of a class are maintained in a ClassHistory object. A ClassHistory is essentially an array of classes intended to be versions of one another. You can place classes in the same ClassHistory in a number of ways. One way is to define a new class with the same name as an existing class. The underlying system will recognize the common name and automatically place the new class as a version of the old class. Another way is to use a variation of the standard class creation message in which you can specify the old class from which you would like to create a new version. In this way, you can create a version of a class in which the new class does not have the same name as the old class. In either case, each class references a ClassHistory (possibly shared) that maintains versioning information about the class.

The following example illustrates how to create a new version of a class. I will continue this example to illustrate how to control migration of instances from the old class to the new class. Suppose you initially create the following class to model a Part in a manufacturing application:

```
Object subclass: #Part
    instVarNames: #(partNumber manufacturer)
    classVars: #()
    poolDictionaries: #()
    inDictionary: UserGlobals
    constraints: #[ #[#partNumber, Integer] ]
    isInvariant: false
```

Using Smalltalk since 1986, Jay Almarode has built CASE tools, interfaces to relational databases, multi-user classes, and query subsystems. He is currently a Senior Software Engineer at GemStone Systems Inc., and can be reached at almarode@slc.com.

A few things in this example are different from the class creation messages in single-user Smalltalks. First, there is the "inDictionary:" keyword used to specify in which symbol dictionary you would like to place this new class. This is used for name space management—GemStone Smalltalk provides a scoping mechanism in which names can be resolved from multiple dictionaries, possibly shared by multiple users. Second, the "constraints:" keyword allows you to specify the kind of object that can be stored into an instance variable. You are not required to specify the constraint for an instance variable, but it is useful for data consistency as well as for speeding up queries. Finally, the "isInvariant:" keyword is used to specify that once an object of this class is committed, no other changes can occur to it. In this example, the object is not invariant, so users can modify instances.

One of the nice things about Smalltalk is the dynamic nature of application development. You can easily modify a class definition and test the new code immediately.

Now suppose that after building an application, you find out that the class definition above is insufficient. You learn that part numbers may contain alphabetic characters, so they should be modeled as Strings, not Integers. Also, you realize that for performance reasons, a Part should also maintain its cost, rather than looking it up from the manufacturer. This leads you to create the following new version of class Part:

```
Object subclass: #Part
    instVarNames: #(partNumber manufacturer cost)
    classVars: #()
    poolDictionaries: #()
    inDictionary: UserGlobals
    constraints: #[ #[#partNumber, String] ]
    isInvariant: false
```

Because this new class has the same name, the new class is automatically created as a new version of the old class. Because you can send the message "classHistory" to the new class to get its ClassHistory object, you can execute "Part classHistory at: 1" to get the old class (because you can no longer reference the old class by the name #Part). Remember, the ClassHistory object is like an array of classes, so the old class is stored in the first slot, and the new class is stored in the second.

Because there is a new class Part that defines instances to be structurally different than the original class, at some point you may want to migrate the instances of the old class to become instances of the new class. The first thing you must do is tell the old class to which class you would like to migrate its instances. This is done by sending the message "migrateTo:" to the old class. For example, if you stored a reference to the old class in the global name #OldPart, then you would execute "OldPart migrateTo: Part." This allows you to migrate instances of a class to any other class, as long as they reside in the same class history. If you forgot to create a new class as a version of an existing class (this could happen if the new class has a different name), you can always add a class to another class's history by sending the message "addNewVersion:" to the old class with the new class as the argument.

Once the migration destination has been set, you can migrate an instance of the old class by sending it the "migrate" message. The default implementation will map the values of instance variables as you might expect. If the new class has an instance variable with the same name as the old class, then the value of that instance variable will be retained. Any new instance variables will have a value of nil. If desired, you can override the method that assigns values to instance variables during migration. This allows you to assign the value of an instance variable that is named differently, or to initialize the value of a new instance variable to something other than nil. In our example, we added a new instance variable to class Part. If we want to initialize the "cost" instance variable, we could override the method as follows:

```
Method for Part
migrateFrom: originalPart instVarMap: instVarMapArray

"Initialize the state of the receiver (a instance being
migrated), based upon the state of the original. The
<instVarMapArray> is an array of offsets that associates
instance variables of <originalPart> to the receiver."

"Handle same-named instance variables the usual way"
super migrateFrom: originalPart instVarMap: instVarMap.

"Lookup the cost from the manufacturer and assign it to
the receiver"
self cost: (self manufacturer costForPart: self partNumber)
```

Our redefinition of class Part has also introduced a subtle problem. Although both the old and new definitions of Part have an instance variable called #partNumber, the new definition has constrained the part number to be a String. An attempt to migrate an old instance will result in an error when an integer is forced into an instance variable that must store a string. This problem is easily solved by overriding the appropriate method in the old class. This method is invoked when the constraint violation occurs and should return the new value desired for the instance variable. Here is a possible implementation of the method:

```
Method for OldPart
invalidInstVarConstraintWhenMigratingInstVar: instVarName
shouldBe: aClass

"The receiver could not be migrated due to having an
instance variable <instVarName> whose value is not a
```

continued on page 27

Hardware

Alan Knight

There are certain recurring ideas in the Smalltalk community. One of these is that we will one day have special hardware that can run Smalltalk as fast, or faster, than C. For example, Stefan Monnier (sm86+@andrew.cmu.edu) writes:

> **I believe that speed increase will mainly become possible with special hardware. Since current processors are specifically designed to run C-like programs, it seems normal that different language paradigms can't really [compete].**

Is this true? Certainly the standard processors have been optimized for traditional languages. If Smalltalk is going to compete against this advantage, maybe it needs special hardware. We'd have Smalltalk-specific instructions, and maybe a garbage collection co-processor to go with it.

BENCHMARKS

Before we even get into CPUs, we should define what we're aiming for. For a real application, how much slower do we expect Smalltalk to be? Although Smalltalk can do very poorly on simple benchmarks, the typical simple benchmark is just a loop doing arithmetic operations. It does no memory allocation, no procedure calls, no data structure manipulation, and may well fit entirely in the cache. While the performance numbers for such a program may be interesting, they don't mean much for real-world programs.

Smalltalk suffers on these benchmarks because there's a lot of infrastructure built into the language. Dynamic binding, safety checking, garbage collection, and being "objects all the way down" all have performance costs that far outweigh their benefits on a trivial benchmark. They are there because they have significant benefits for large system development. In my opinion, this kind of infrastructure is important, if not essential, for a wide range of systems. If you don't have it, you'll end up building it, probably much less efficiently than if it were built in.

Unfortunately, "a wide range of systems" is not all systems. I know I've worked on systems where the overhead of dealing with every floating-point number as an object with full dynamic binding would be absolutely fatal.* Let's be ambitious, and try to make Smalltalk workable for all systems. We want a Smalltalk processor that can run Smalltalk just as fast as C runs on a conventional CPU. Besides, all that "real applications" stuff, valid as it is, sounds too much like the marketing drivel that's used to hide a slow implementation.

OBSTACLES

We've got our hypothetical processor in mind. Let's call it SKAMP (for Smalltalk Kick-Ass Mega-Processor). What do we need in order to build it?

The biggest obstacle is money, which is directly related to economies of scale. Creating a fast custom processor is a very expensive business. To keep the price/performance ratio acceptable, we'll need to sell a lot of units. This is much easier for a general-purpose CPU than one tuned for a particular language. Maybe there's someone else we can base our business model on. There haven't been any commercial Smalltalk machines, but we can use something quite similar, LISP machines. Unfortunately, that's not a very inspiring example. John Nagle (nagle@netcom.com) writes:

> **That was the premise behind Symbolics and their LISP machines. It turned out, though, that simple RISC machines with a decent compiler run Common LISP with substantially better price/performance than Symbolics' refrigerator-sized machines.**

and Markus Stumptner (mst@vexpert.dbai.tuwien.ac.at) writes:

> **[It] is not that such an implementation is impractical at a given time, but that, quite generally, while research projects may initially produce fast designs, there just is not enough money behind language-specific processors to keep pace with the speed increases of general purpose machines. I don't think this is going to change.**

Even if we can make enough money to stay in business producing this chip, we run into issues of compatibility

Alan Knight is an adaptive neural network with hardware and software optimized for semantic processing and connected via a speech recognition interface to The Object People, 885 Meadowlands Dr., Ottawa, Ontario, Canada, K2C 3N2. He can be interfaced to at 613.225.8812 or as knight@acm.org.

* This does not include financial systems. Anybody that's even thinking of using floating-point for financial calculations should go and read my previous columns on math (The Smalltalk Report 4[7] and 4[8]).

JUST PUBLISHED!

Rapid Software Development with Smalltalk

The Ultimate Guide to Better Smalltalk Development...Write Code Faster Without Sacrificing Quality.

RAPID SOFTWARE DEVELOPMENT WITH SMALLTALK covers the spectrum of O-O analysis, design, and implementation techniques and provides a proven process for architecting large software systems. By using detailed examples of an extended Responsibility-Driven Design (RDD) methodology and Smalltalk, readers will find techniques derived from real O-O projects that are directly applicable to on-going projects of any size.

The author provides readers with specific guidelines that could dramatically cut costs and keep projects on-time. Specifically, the author provides readers with project patterns that work, illustrations of design patterns, O-O metrics with example code to test design quality and of course, numerous Smalltalk code examples.

Readers will...

- Speed up the development process by fostering reuse
- Significantly reduce debugging time
- Gain step-by-step instruction on how to make the object model more robust
- Learn how to distribute responsibilities within the object model more effectively
- Discover a practical day-by-day breakdown of a rapid modeling session
- See how to organize the development team most efficiently

This book will prove invaluable to anyone interested in speeding up the consistent development of high-quality object-oriented software systems based in Smalltalk.

About the Author...

Mark Lorenz is the founder and president of Hatteras Software, Inc. a company that specializes in helping projects use object technology successfully. The author has already published two popular books on object technology entitled OBJECT-ORIENTED SOFTWARE DEVELOPMENT: A PRACTICAL GUIDE and OBJECT-ORIENTED SOFTWARE METRICS (Prentice Hall) and also writes a regular column for THE SMALLTALK REPORT called "Project Practicalities."

ADVANCES IN OBJECT TECHNOLOGY

Available at selected book stores.
Distributed by Prentice Hall.

SIGS BOOKS ORDER FORM

❑ **YES!** Please rush me __ copy(ies) of RAPID SOFTWARE DEVELOPMENT WITH SMALLTALK at the low price $24 per copy. (ISBN: 1-884842-12-7; Approx 200 pgs.)

Money-back Guarantee: If I am not completely satisfied, I may return the book(s) within 14 days and receive a complete refund, promptly and without question.

❑ Check payable to SIGS Books
❑ Visa ❑ American Express ❑ MasterCard

Card# ________ Exp. ____

Signature ________

SHIPPING AND HANDLING: For US orders, please add $5 for shipping and handling; Canada and Mexico add $10; Outside North America add $15. IMPORTANT: NY State residents add applicable sales tax. Please allow 4–6 weeks for delivery.

Name ________
Company ________
Title ________
Address ________
City/State/Zip ________
Country/Postal Code ________
Phone ________ Fax ________

SEND TO:
SIGS Books, P.O. Box 99425
Collingswood, NJ 08108-9970
Phone: 609.488.9602 Fax: 609.488.6188

SIGS BOOKS

and reuse. It's possible for Smalltalk to perform the functions of an operating system, and early implementations did just that. The question is whether it's feasible, or even desirable in today's environment.

Suppose we've got our Smalltalk on a chip, running as its own operating system. How many different video cards will it support? Can it use VBXs? Will it run MS Word, or do we also have to write all our own applications? Maybe SKAMP will be so compelling that all the applications developers will immediately start writing for it. Maybe not.

We, the champions of OOP, should be practicing reuse, not reinventing the wheel. Smalltalk is a programming language and a development environment. It shouldn't also have to be an operating system and every application you'd ever want. Let people who understand operating systems write the operating system, and those who understand application development develop applications. I want an environment that can take advantage of their work, not one that forces me (or the Smalltalk vendors, who seem to have enough trouble getting their own environments right) to duplicate it.

That's my attitude, and I'm a confessed Smalltalk bigot. The average user will be much less sympathetic to the purity and elegance of the environment if it means they have to buy a special machine that won't run their favorite application.

The typical simple benchmark does no memory allocation, no procedure calls, no data structure manipulation, and may well fit entirely in the cache.

DORADO AND SOAR

OK, we'll admit there are a few obstacles, but surely sufficiently good performance could overcome at least some of them. Hasn't anybody even tried Smalltalk hardware?

Yes, they have. In fact, Smalltalk's first implementation was on hardware built at Xerox PARC for that purpose. They may not have had Smalltalk CPUs, but they did have Smalltalk-specific microcode. These evolved into the Dorado, a highly optimized Smalltalk engine that remained the Smalltalk reference machine for many years. The ParcPlace Advanced Tools benchmarks still rate performance in terms of a Dorado. The machine was described as:

> This 70 ns ECL minicomputer costs $120,000 [*in 1985*] and dissipates over 2 kilowatts, requiring an air-conditioned room.[1]

In the mid-1980s, a group at Stanford University created SOAR, a RISC chip optimized for Smalltalk. At the time, RISC chips were a relatively new thing, and unproven for languages like Smalltalk. This group managed to create a Smalltalk-specific RISC chip that had roughly the same performance as a Dorado despite having a cycle time that was five times slower. This showed very clearly that a RISC machine could run Smalltalk well, and in fact it showed that not many additional features were necessary. David Ungar[1] describes the important features of SOAR:

> The most important hardware features are register windows and tagged integer instructions. These two features nearly double SOAR's performance by reducing the cost of subroutine calls and type-checked integer operations...

This was a very impressive result in Smalltalk-specific hardware. Why wasn't it carried forward into commercial hardware? There's a simple answer. It was. SOAR is a direct ancestor of Sun's SPARC chips, which also support register windows and tagged integer operations. Those of you with SPARCstations are already using machines with features to improve Smalltalk performance.

COMPILERS

The information on SOAR is from David Ungar's doctoral thesis. Later, he led a research group working on an optimizing compiler for Self, a Smalltalk-like language that is well known in the Smalltalk community for its impressive benchmarks, running at approximately half the speed of optimized C, even on integer manipulation benchmarks, on which Smalltalk does very badly.

Self is a continuing research project, with many publications. Information, and copies of some of the papers can be obtained by ftp from self.stanford.edu or over the World Wide Web from http://self.stanford.edu.

Urs Hölzle (urs@cs.stanford.edu), a member of the Self project, writes:

> **In my experience (which is based on several years of writing optimizing compilers for Self...) there is very little to be gained from special hardware *IF* you have an optimizing compiler (which no Smalltalk system has today, as far as I know).**
>
> **To be more concrete, I recently measured the instruction-level behavior of optimized Self programs, and I could not find any single hardware feature that would improve performance by more than 5%. The hardware "feature" with the biggest impact was cache size, and its impact was larger than all of the "OO" features (tagged arithmetic, register windows, a hypothetical 1-cycle "lookup" instruction) combined. So I would claim that a standard RISC + reasonably large cache will run *optimized* Smalltalk programs faster than any "Smalltalk-in-Hardware" chip built with the same chip technology. In other words, special hardware isn't a loser just for market size reasons, it also loses on the performance side.**

From this point of view, it seems there's a lot more to be gained by pestering the Smalltalk vendors for better compilers than in pushing for language-specific hardware. Even here there are a few caveats. Self-style optimizations

can have significant space costs that may make them inappropriate for some environments. Also, those of us using low-end systems don't have an optimizing compiler *or* a RISC CPU, and we're lucky if we get to use a 32-bit operating system.

THAT'S NO FUN

This is pretty discouraging news for those who were looking to Smalltalk hardware for performance improvements. It doesn't mean there's no hope, but the possibility looks pretty remote. If special-purpose hardware is going to succeed, it's seems to me it will have to be a much more radical departure, perhaps in the form of massively parallel object systems. In the short term, it looks like we're going to have to rely on better compilation techniques, conventional hardware improvements, and our own ability to use efficient algorithms. We're also going to have to keep explaining that bit about benchmarking real applications, even if it does make us sound like marketing people.

Reference

1. Ungar, D.M. The Design and Evaluation of a High Performance Smalltalk System, MIT Press, Cambridge, MA, 1987.

GETTING REAL

continued from page 23

kind of <aClass> (the constraint in the migration destination). This method performs a conversion to return a new value of kind <aClass>."

```
instVarName = #partNumber
    ifTrue: [ ^ self partNumber asString ]
    ifFalse: [
        ^ super
            invalidInstVarConstraintWhenMigratingInstVar:
instVarName
            shouldBe: aClass
    ]
```

There are a number of strategies for when to perform instance migration. One strategy is to migrate all instances of the class at once. This requires a scan of the entire object memory to collect all instances, then invoking the message to migrate each instance. When an object is migrated, it is written, so there is a chance of concurrency conflicts if other users are accessing the same instances. You can acquire locks to make sure the migrations can be committed, but this reduces availability for other users. Migrating all instances is best performed when the system is relatively inactive or when there are no other users accessing these objects. Another strategy is to migrate instances when they are accessed by an application. By performing "migration on demand," the semantics of the application can determine whether to perform the migration or not.

INFO@SIGS

SIGS Publications, Inc., 71 West 23rd Street, 3rd Floor, New York, NY 10010; 212.242.7447; Fax: 212.242.7574

ARTICLE SUBMISSION

To submit articles for publication, please contact: John Pugh & Paul White, Editors, 885 Meadowlands Dr. #509, Ottawa, Ontario, K2C 3N2 Canada; email: streport@objectpeople.on.ca

PRODUCT REVIEWS AND ANNOUNCEMENTS

To submit product reviews or product announcements, please contact the Editors at the address above.

CUSTOMER SERVICE

For customer service in the US, please contact PO Box 5050, Brentwood, TN 37024-5050; 800.361.1279; Fax: 615.370.4845; in the UK, please contact Subscriptions Department, Tower Publishing Services, Tower House, Sovereign Park, Market Harborough, Leicestershire, LE16 9EF, UK; +44.(0)1858.435302; Fax: +44.(0)1858.434958

SIGS BOOKS

For information on any SIGS book, contact: Don Jackson, Director of Books, SIGS Books, Inc., 71 West 23rd Street, New York, NY 10010; 212.242.7447; Fax: 212.242.7574; email: donald_jackson@sigs.com

SIGS CONFERENCES

For information on all SIGS Conferences, please contact: SIGS Conferences, 71 West 23rd Street, 3rd Floor, New York, NY 10010; 212.242.7515; Fax: 212.242.7578; email: info@sigs.com

BACK ISSUES

To order back issues, please contact: Back Issue Order Department, SIGS Publications, 71 West 23rd Street, 3rd Floor, New York, NY 10010; 212.242.7447; Fax: 212.242.7574

REPRINTS

For information on ordering reprints, please contact: Reprint Management Services, 505 East Airport Road, Box 5363, Lancaster, PA 17601; 717.560.2001; Fax: 717.560.2063

ADVERTISING

For advertising information, please contact: Advertising Department, SIGS Publications, 212.242.7447; Fax: 212.242.7574

SIGS HOME PAGE

To access the SIGS Home Page on the World Wide Web: http://www.sigs.com.

Merging Smalltalks: THE SMALLTALK REPORT talks to Jim Anderson, Co-Chairman of ParcPlace-Digitalk

As most of you are aware, Digitalk and ParcPlace have recently merged to become ParcPlace-Digitalk. Digitalk and ParcPlace marketed the Visual Smalltalk and VisualWorks product lines, respectively, mainly to the commercial development organizations in large corporations. We've asked Jim Anderson, one of the founders of Digitalk and now Co-Chairman of ParcPlace-Digitalk, to discuss product strategy issues of concern to Visual Smalltalk and VisualWorks users.

What was the pre-merger marketing situation?
Large companies have been adopting Smalltalk for their business applications because it deals with change and complexity so well. Frequently companies retrain COBOL programmers to be Smalltalk developers. About one-third of the time, ParcPlace and Digitalk have been competing with each other for the same customers. Often these customers are attempting to standardize on a single Smalltalk vendor. The strengths of our two products are so different that choosing one over the other has been a difficult choice.

What differentiates the two products?
VisualWorks features extraordinary portability and platform coverage. Both its Smalltalk image and its underlying C-implemented virtual machine have demonstrated a depth of portability unmatched by any other development tool. The breadth of platforms covered, 12 in all, is also impressive, including Windows 3.1, NT (Intel and DEC Alpha), the Macintosh (68K and PowerPC), OS/2, Sun Solaris, HP UX, and AIX. In addition, VisualWorks has proven to be scalable to extremely large corporate applications. Both VisualWorks and Visual Smalltalk are the leaders in Smalltalk execution performance.

Visual Smalltalk features extensive platform fidelity, a component assembly paradigm, group development support, and runtime modularity. It supports the look and feel of Windows and OS/2, and the soon to be released Version 3.1 has already received Windows 95 logo certification. It includes the PARTS Workbench for assembling components into applications by "wiring" instead of writing code. Visual Smalltalk Enterprise includes the Team/V group development tools and API (for user-configurable/created and our own vendor-supplied tools). Visual Smalltalk supports runtime modularity in the form of Smalltalk Link Libraries (SLLs). These provide footprint control, cross-application sharing, and incremental maintenance replacement.

What are the plans for upcoming releases post-merger? Will both product lines continue to be supported?
In the fourth quarter of 1995 we plan to ship VisualWorks Release 2.5 and Visual Smalltalk Release 3.1, as previously announced. These releases will feature 100% language syntax and semantics compliance between the two products, which conforms with the draft ANSI Smalltalk specification from the X3J20 committee in which both companies participate. The languages are already quite similar, with minor differences in literal arrays and blocks that affect very few customers.

Concurrent with these releases, we will issue a document on base class commonality to guide customers in creating portable business objects. The base classes are also quite similar. We have several customers today that use Visual Smalltalk on the client working with VisualWorks on the server. At the ParcPlace and Digitalk International Users Conference in August, we demonstrated such a customer application. It was created as a client-only Visual Smalltalk application. It was then partitioned into presentation objects on the Visual Smalltalk client, and business objects on the VisualWorks server, at a development cost of less than two person-weeks. The majority of the two weeks were spent re-architecting an application intended to run in fat-client mode to a client and server model. Most of the business logic moved over without modification.

In 1996, we plan to bring the best of both products for-

continued on page 32

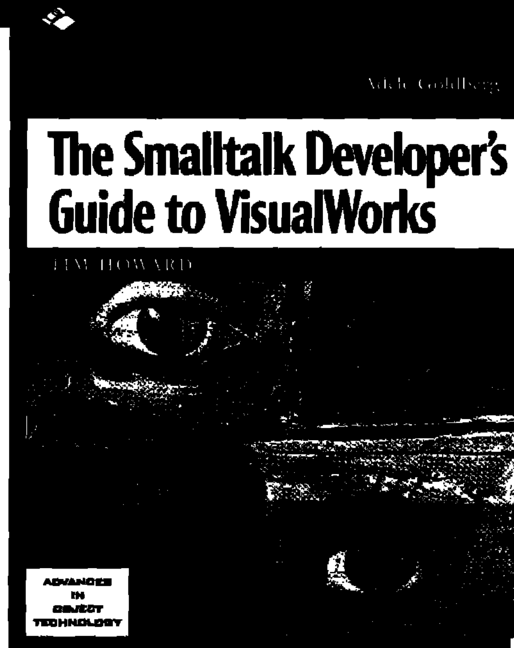
BONUS: Diskette included with each copy

See VisualWorks Come Alive with this Complete New Guide

The Smalltalk Developer's Guide to VisualWorks

BY TIM HOWARD

Foreword by Adele Goldberg



THE SMALLTALK DEVELOPER'S GUIDE TO VISUALWORKS provides an in-depth analysis of the popular application development tool produced by ParcPlace Systems. Designed to enhance development acumen, this book serves as a guide to using VisualWorks to its full potential.

Divided into two logical parts, the reader first receives the basic principles of VisualWorks and then is provided with concrete examples of VisualWorks in action. In this way, you are sure to gain a better understanding of the unique characteristics of this powerful development tool as well as a complete understanding of its strengths and weaknesses. By reading this book, you'll be able to build better applications and enhance the tools themselves.

And as an added bonus, source code and numerous examples of the outlined concepts are provided on the included diskette. You'll be able to test the concepts immediately and put theory into practice as you read.

If you are a professional software developer already programming in VisualWorks or an advanced Smalltalk programmer, this book will prove an invaluable guide to enhancing your skills, cutting development time, and saving money.

Not recommended for beginning programmers.

Available at selected bookstores.
Distributed by Prentice Hall.

SIGS ISBN: 1-884842-11-9
PH ISBN: 0-13-442526-X
Diskette included

PART OF THE ADVANCES IN OBJECT TECHNOLOGY SERIES

Complete and easy to read, you can use this book as:

- a study guide
- a series of tutorials
- a reference for items and concepts
- a valuable source of VisualWorks code

Eminently useful, this book is unique because:

- Each topic is reinforced with a concrete example. The concepts are clearly illustrated and the reader can actually see their application.
- A special browser is provided containing all the examples referenced, alleviating the need to enter code.
- Rigorous definitions of terms are provided to mitigate confusion.
- Applications built *prior* to VisualWorks are covered to build an understanding of where some of the constructs in VisualWorks originated.
- Detailed descriptions of how to add new components to the palette are illustrated, allowing the reader to extend the functionality of VisualWorks. Three new components are provided as examples.

SIGS BOOKS ORDER FORM

❑ **YES!** please send me _____ copy(ies) of THE SMALLTALK DEVELOPER'S GUIDE TO VISUALWORKS at the low price of $39 *(diskette included)*
ISBN: 1-884842-11-9. Approx. 630 pages.
Money Back Guarantee: If I am not completely satisfied, I may return the book(s) within 14 days and receive a complete refund, promptly and without question.

Method of Payment

❑ Check Enclosed *(payable to SIGS Books)*
❑ Charge My: ❑ Amex ❑ MasterCard ❑ Visa

Card # ______________________ Exp______

Signature ______________________

Shipping & Handling: For US orders, please add $5 for shipping & handling, Canada & Mexico add $10, outside N. America add $15. NY State residents add applicable sales tax. Please allow 4-6 weeks for delivery.

Name ______________________

Title ______________________

Company ______________________

Address ______________________

City ______________ State _____ Zip ______

Phone/Fax ______________________

SEND TO:
SIGS Books, P.O. Box 99425
Collingswood, NJ 08108-9970
Fax To: 609-488-6188
Phone: 609-488-9602

SIGS BOOKS

KNOWLEDGE SYSTEMS CORPORATION

Make No Compromises. Join a leader in Object Technology.

We are Knowledge Systems Corporation, the acknowledged leader in Object Oriented Technology services. Working on the cutting edge of technology, we are poised to move to greater heights of technical diversity, client serviceability, and employer opportunity. We are professional, team oriented, and driven to excellence, but most of all, we are an employee-oriented corporation that provides an excellent working environment that will challenge your abilities and sharpen your skills. *We are KSC. We are your future.*

Presently, we are seeking to augment our technical training and consulting staffs with professionals who have two plus years of demonstrated experience with OOA&D, IBM Smalltalk or VisualAge, ParcPlace VisualWorks, Digitalk Smalltalk/V, and ENVY/Developer.

As a leader in supplying our Fortune 500 client base with Object Oriented solutions, Knowledge Systems Corporation is able to offer a very competitive salary, an excellent benefits package and many opportunities to grow with the leader. Please send/fax your cover letter, resume, and salary requirements to: Knowledge Systems Corporation, 4001 Weston Parkway, Cary, NC 27513; or call (919) 481-4000; Fax (919) 677-0063 or e-mail to jdemichiel@ksccary.com.. Equal Opportunity Employer.

KNOWLEDGE SYSTEMS CORPORATION

SMALLTALK POSITIONS

ParcPlace-Digitalk is seeking experienced Smalltalk instructors and consultants for our world-class Professional Services team. At **ParcPlace-Digitalk** you will work with one of the world's leading development teams, use state-of-the-art products and assist companies on the forefront of adopting object technology in client-server applications.

Requirements for Senior Consultants: solid experience with Smalltalk (3-5 years) and/or PARTS Workbench experience. OOA/D experience and GUI design skills. Mainframe database experience is a big plus. Requirements for instructors: previous training experience in a related field (2-4 years), understanding of OO concepts and Smalltalk.

Positions are available in various sites throughout the U.S. Compensation includes competitive salary, bonuses, equity participation, 401(k) and family medical coverage. All positions require travel. **ParcPlace-Digitalk** is an equal opportunity employer.

Please forward your resume to:
Director of Enterprise Services
ParcPlace-Digitalk, 7585 S.W. Mohawk Drive
Tualatin, OR 97062 fax: (503) 691-2742
internet: holly@digitalk.com

Smalltalk and C++ Experts
30 IMMEDIATE OPPORTUNITIES
Chief Architects • Instructors • Mentors

ObjectSpace, a leader in **the Object-Oriented arena,** has enjoyed 300% growth in the last year, and as a result, has IMMEDIATE opportunities for extraordinarily talented people dedicated to the creation and deployment of advanced technologies. Our areas of interest include: CORBA, OODBMS, Constraint-based Programming, Rule-based Programming, Prototype-based Languages (Classless), as well as Agent Technology, Design Patterns, Biological Systems, Cognitive Science, OOA/OOD and Self.

Our requirements for **EXPERTS** committed to excellence include 4+ years of experience with C++, Smalltalk, Distributed Smalltalk, VisualWorks or VisualAge. In addition, candidates should also possess expertise in Object-Oriented Software Development Methodologies.

We offer competitive compensation, performance-based bonuses and a complete benefits package. For immediate consideration, forward your resume to:

Fax (214) 663-9099

ObjectSpace, Inc., 14881 Quorum Dr., Suite 400, Attn: ST1095, Dallas, TX 75240; jobs@objectspace.com; or call (800) OBJECT1. EOE.

SMALLTALK *Opportunities*

Computer Professionals, Inc., an IBM Business Partner and award winning consulting firm, has immediate openings in Florida for Smalltalk professionals with 1-5 years of experience. Tremendous opportunity to expand your object oriented career development. Knowledge of ENVY or GEMSTONE is a plus, but is not necessary.

CPi specializes in client server, object oriented technologies and offers access to a large Fortune 1000 client base. CPi offers a competitive salary and benefits package with large growth potential.

For more information, forward resumes or call:
CPi
12360 66th Street North
Largo, FL 34643
TEL: 800/257-7308 FAX: 813/224-9144
E-Mail to cpi7@occ.com EOE

Australia **Smalltalkers Wanted!** Asia

Browse Classes

File Smalltalk Classes

Person
Point
Rectangle

address
age
ambitions
careerMove

```
careerMove

(self hasExtensiveExperienceIn: Smalltalk)
  & (self canAccept: #bigChallenges in: highProfileProjects
    locatedThroughout: #(Australia Asia) )
      ifTrue: [self respondTo: 'bobhall@vnet.ibm.com']
      ifFalse: [self stayPut]
```

We seek first rate experienced Smalltalk software professionals, to be part of leading edge Object Oriented projects in the finance industry in Australia and throughout Asia.

We have can offer careers or term contracts, and have positions available that include developers, team leaders, mentors and educators.

Contact Bob Hall, IBM Australia
fax: 61.2.353.3604
internet:bobhall@vnet.ibm.com

RECRUITMENT CENTER

To place an ad in this section, call Michael Peck at 212.242.7447

EDITORIAL

continued from page 2

All the major Smalltalk vendors have special packages for educational institutions that make it easier for departments. Be aware that the hardware requirements might be more demanding than what your department has available, not to mention students' own machines at home. Third, try introducing Smalltalk into your data structures course first, rather than your programming languages course—we certainly had great success with this approach. Fourth, try not to compromise in terms of how objects get introduced. Often, the issue gets brushed off by the statement "we teach C++," which is fine if the statement is true, but more often than not what is taught is C using a C++ compiler.

We would like to thank Jim Anderson for allowing us to discuss with him the future of the new ParcPlace-Digitalk venture. Jim, of course, was one of the founders of Digitalk and has been a true champion of Smalltalk from its early commercial days. We plan to use this interview as a launching pad for introducing you to other players within the Smalltalk arena. We hope you will enjoy this new feature in coming months.

INTERVIEW

continued from page 28

ward in a new integrated product. Incremental release of this new product is targeted to begin in the second quarter.

We will start with the portable VisualWorks virtual machine as the basis, extended with the Visual Smalltalk SLL capability. Added to this will be the VisualWorks base classes and portable GUI framework, as well as the Visual Smalltalk event framework. Visual Smalltalk platform integration will be added, consistent with the portable GUI framework, for selected platforms. Initially, all Windows platforms and OS/2 will be supported. This will be accomplished using SLL technology to plug in a platform-integrated implementation dynamically in place of the portable implementation. We will also add the Visual Smalltalk team API and infrastructure. On top of this will be hosted the tools from both Visual Smalltalk and VisualWorks, including the canvas, browsers, and data lens from VisualWorks, the PARTS Workbench, and Team/V tools from Visual Smalltalk.

Which product should we buy today to be best aligned with this new product direction?
That depends on your priorities. If platform portability or UNIX platforms are very important for you today, then begin with VisualWorks. On the other hand, if platform fidelity, Windows 95 compliance, or component-based development using OCX controls are very important for you today, then begin with Visual Smalltalk. Whichever choice our customers make, we'll ensure they have a migration path toward the new products.