The Smalltalk Report

The International Newsletter for Smalltalk Programmers

July/August 1992 Volume 1 Number 9

Implementing Peer Code Reviews in Smalltalk

By S. Sridhar

Contents:

Features/Articles

1 Implementing Peer Code Reviews in Smalltalk *by S. Sridhar*

3 Quality assurance issues for Smalltalk-based applications *by Ed Klimas*

Columns

14 *The Best of Comp.Lang.Smalltalk:* What's wrong with OOP? *by Alan Knight*

18 *Smalltalk Idioms:* Abstract control idioms *by Kent Beck*

21 *Getting Real:* Creating subclasses *by Juanita Ewing*

Departments

24 *Lab Report:* Smalltalk research at the University of Florida *by Justin O. Graver*

25 *Product Announcements*

26 *Highlights*

Smalltalk programmers routinely read more code than they write. One of the more difficult things in mastering Smalltalk is mastering the underlying class libraries. The act of programming consists of "hunting" for the appropriate protocols in the appropriate classes. The hallmark of a good Smalltalker is the ability to efficiently navigate the increasingly vast landscape of the class libraries. It is very likely that a well-written class will be read many times over. Typically, these classes serve as role models for people who aspire to be good Smalltalkers. In the same vein, badly written classes that have to be used all the time serve as poor role models and tend to propagate poor programming practices.

Code reviews are a critical software engineering activity. Research studies have shown that code inspection alone can catch 75–90% of the errors. Code reviews serve as a critique of a software component by someone other than the author of the component. Typically, reviewers are peers in the same organization who may or may not be on the development team. Different organizations employ different practices for implementing code reviews. In traditional software organizations, the author prints out a listing of the modules to be reviewed and circulates it to the review team. The reviewers mark it up using guidelines. They meet with the author in a face-to-face meeting a few days later where they present and discuss the review comments page by page. The author is then left with the daunting task of collating and integrating the comments of all the reviewers. It is quite possible that more than one person has caught the same programming infraction; that means multiple reviewers have spent time documenting the same problem and possibly suggesting the same fixes. This is an unnecessary duplication of effort.

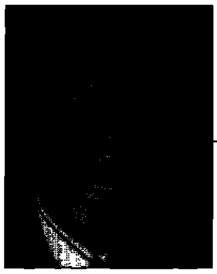
This article describes a code review process that is specifically geared to reviewing small or large chunks of Smalltalk code. I'll discuss extensive guidelines for effective code critique and review, and describe a rigorous process that we have adopted in our organization to execute a comprehensive review.

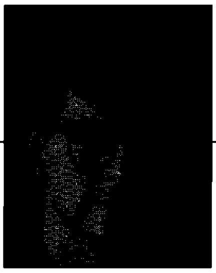
THE PROCESS

My work group, like most Smalltalk work groups, uses a team software engineering tool to manage the software development process. The tool we use is the ENVY/Developer environment (hereinafter referred to as ENVY). This tool provides the facilities required for the development and maintenance of large software systems. It provides sophisticated versioning, configuration and release management capabilities as well as mechanisms for co-ordinating and integrating the software development activities of multiple developers. All of the program development information including source code, compiled methods, module dependency graphs, component ownership, versioning and configuration details are stored in a shared repository. This repository is accessible to all members of the technical team. We use ENVY as a practical tool to aid in our code review process. It would be feasible to adapt this approach to code reviews in the context of other team development environments as well.

At the outset of a project, we typically partition the tasks among the developers along functional lines. For example, one developer may be working on specific domain abstrac-

continued on page 8...


John Pugh


Paul White

EDITORS' CORNER

"Reuse is the name of the game. Headlines shout. Marketing literature trumpets. Salesmen ooze. Objects will solve your reuse problems. Not true, of course. Programmers solve reuse problems." So says Kent Beck in the prelude to his Smalltalk Idioms column in this issue. Kent knows, as do most experienced Smalltalk programmers, only too well that reuse does not come for free and is far from easy to attain. Developing truly reusable components adds an extra dimension to the design and programming process, and additional time and skill is required.

What if we had access to a quality set of reusable components for objects for some application domain? Would it now be easy to "assemble" new applications from these components? Unless a great deal of thought has gone into the design and implementation of the components (probably not), we will still need a Smalltalk guru to wire the pieces together. Here is a sample list of questions for which we would need affirmative answers. Do we have the right components? Do the components have the required functionality? Have they been fully tested? Do we have prefabricated components that model common processes within the domain, or do we simply have a collection of low-level building blocks? Have the components been designed with standard interfaces? Have they been designed to work together?

Clearly, a number of things have to be in place before we can use a software-by-assembly approach to application development. However, there are lots of analogous real-world examples where the benefits of this approach can be clearly seen such as the assembly of personal computers or the manufacture of automobiles.

It's this software-by-assembly approach to application development that Parts (formerly LAFKit—Look and Feel Kit), a product currently under development at Digitalk, is attempting to address. In his keynote address at Object Expo in New York City, Digitalk CEO Jim Anderson teased the audience with a preview of the notions of software construction from parts. Watch for more information on Parts in future issues of *The Smalltalk Report.*

This issue features two significant contributions from recognized members of the Smalltalk community that deal with the delivery of quality software. First, S. Sridhar returns to *The Smalltalk Report* with a description of a mechanism for carrying out peer code reviews on Smalltalk projects. As is the case with all software development, whether done using Smalltalk or any other language and environment, code reviews are a vital to ensuring the quality of deliverables. As Sridhar points out, "code reviews should be viewed as a rigorous software enginering activity." His discussion focuses on two aspects of code reviews, namely the process by which they should be carried out and a set of guidelines to be followed by reviewers. Second, Ed Klimas' article discusses many issues that make testing Smalltalk systems unique and puts forward a number of guidelines and strategies for planning for and documenting the development of test suites.

Also in this issue, Juanita Ewing addresses many of the problems involved in carrying out proper subclassing within Smalltalk applications. Always a confusing issue for people new to Smalltalk, she describes the differing goals subclassing is used to satisfy and proposes hueristics for making decisions concerning subclassing decisions. Alan Knight brings us up to date on a thread of discussion on USENET dealing with the hype surrounding object technology. And finally, Justin Graver describes the activities of the Smalltalk research group at the University of Florida.

John Pugh [illegible]

The Smalltalk Report (ISSN# 1056-7976) is published 9 times a year, every month except for the Mar/Apr, July/Aug, and Nov/Dec combined issues. Published by SIGS Publications Group, 588 Broadway, New York, NY 10012 (212)274-0640. © Copyright 1992 by SIGS Publications, Inc. All rights reserved. Reproduction of this material by electronic transmission, Xerox or any other method will be treated as a willful violation of the US Copyright Law and is flatly prohibited. Material may be reproduced with express permission from the publishers. Mailed First Class. Subscription rates 1 year, (9 issues) domestic, $65, Foreign and Canada, $90, Single copy price, $8.00. POSTMASTER: Send address changes and subscription orders to: THE SMALLTALK REPORT, Subscriber Services, Dept. SML, P.O. Box 3000, Denville, NJ 07834. Submit articles to the Editors at 91 Second Avenue, Ottawa, Ontario K1S 2H4, Canada.

The Smalltalk Report

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

SIGS PUBLICATIONS
Advisory Board
Tom Atwood, Object Design
Grady Booch, Rational
George Bosworth, Digitalk
Brad Cox, Information Age Consulting
Chuck Duff, Symantec
Adele Goldberg, ParcPlace Systems
Tom Love, OrgWare, Inc.
Bertrand Meyer, ISE
Meilir Page-Jones, Wayland Systems
Shesa Pratap, CenterLine Software
P. Michael Seashols, Versant
Bjarne Stroustrup, AT&T Bell Labs
Dave Thomas, Object Technology International

THE SMALLTALK REPORT
Editorial Board
Jim Anderson, Digitalk
Adele Goldberg, ParcPlace Systems
Reed Phillips, Knowledge Systems Corp.
Mike Taylor, Digitalk
Dave Thomas, Object Technology International

Columnists
Juanita Ewing, Digitalk
Greg Hendley, Knowledge Systems Corp.
Ed Klimas, Linea Engineering Inc.
Suzanne Skublics, Object Technology Ind.
Eric Smith, Knowledge Systems Corp.
Rebecca Wirfs-Brock, Digitalk

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

Art/Production
Kristina Joukhadar, Managing Editor
Pilgrim Road, Ltd., Creative Direction
Karen Tongish, Production Editor
Jennifer Englander, Art/Prod. Coordinator

Circulation
Ken Mercado, Fulfillment Manager
Diane Badway, Circulation Business Manager
Vicki Monck, Circulation Assistant
John Schreiber, Circulation Assistant

Marketing/Advertising
Diane Morancie, Account Executive
Holly Meintzer, Account Executive
Geraldine Schafran, Recruitment Sales
Sarah Hamilton, Promotions Manager
Caren Polner, Promotions Graphic Artist

Administration
David Chatterpaul, Accounting
Claire Johnston, Conference Manager
Cindy Roppel, Conference Coordinator
Amy Stewart, Projects Manager
Jennifer Fischer, Public Relations
Helen Newling, Administrative Assistant

Margherita R. Monck
General Manager

SIGS PUBLICATIONS

Publishers of Journal of Object-Oriented Programming, Object Magazine, Hotline on Object-Oriented Technology, The C++ Report, The Smalltalk Report, The International OOP Directory, and The X Journal.

QUALITY ASSURANCE ISSUES FOR SMALLTALK-BASED APPLICATIONS

Ed Klimas

It is absurd to divide software into good and bad.
Software is either charming or tedious.
—loosely based upon Oscar Wilde

Smalltalk-based object-oriented programming has several technical advantages for improved programmer productivity and code quality over other programming environments. These include:

- Reuse of existing pretested robust libraries
- Inherent encapsulation against "runaway code"
- The absence of error prone pointer manipulations
- Automatic memory management capabilities

These inherent technical features do not, however, automatically result in higher quality software. It is necessary to take explicit steps to design in quality from the start. One of the most important of these steps is testing.

This article will cover a number of issues associated with commercial Smalltalk application testing and propose a standardized, yet flexible, platform independent protocol so class libraries and frameworks from multiple sources can be easily integrated and tested.

Testing is only part of the quality assurance process. Software quality assurance begins with design and code reviews and progresses through a life cycle that is standard irrespective of the programming tools. Whether a waterfall or iterative development model is employed, the purpose of each step in the software quality life cycle can be summarized by the following stages:

Pre-implementation phase

- Preliminary design reviews
- Detailed design reviews
- Code reviews, inspections and walk throughs

Code development phase

- Unit testing
- Integration testing
- Validation testing

The benefits of pre-implementation code reviews and inspections should not be underestimated. In terms of impacting overall finished product quality, inspections and design reviews are much less costly than computer-based regression testing to isolate the same errors. Many existing review and inspection techniques are quite appropriate to the preimplementation phase of the O-O software quality assurance life cycle. Still, one significant contribution to a commercial product's final quality is the testability of the code.

The object-oriented development paradigm differs from conventional structured development in several ways, and so one should not assume that all O-O testing is necessarily the same. Testing commercial OOP/Smalltalk-based applications still follows the software quality life cycle model. Likewise, Smalltalk does not pose any greater burden than conventional languages in this regard and may in fact offer some advantages under some circumstances. For example, procedural languages such as Fortran usually require several functions to be implemented before testing can begin. Smalltalk programs permit classes to be used and tested as soon as they are designed and the initial underlying methods are defined. This permits meaningful testing to occur earlier in the development cycle and offers the cost saving opportunity to diagnose and correct problems earlier. Therefore, it is important to design Smalltalk programs for easy testability of the code from the start.

REGRESSION TESTING

Unit and integration testing are intended to uncover latent software errors, while validation testing demonstrates traceability to the requirements. Each of these steps typically relies upon some sort of regression testing. Regression testing is an important part of testing the impact of changes on the total body of code. Just as Smalltalk is encapsulated in an environment for development, commercial developers should wrap their products into an environment that will easily support full testability of their work. If a few simple testing guidelines are followed from system conception, significant time can be saved later during the subsequent refinements of the system.

Commercial software requires the development of ancillary test code to verify the proper initial functioning of the software, and to verify that the properly working functions of the software system are not inadvertently altered during subsequent code additions or modifications. Unfortunately, in many organizations, the test suites are usually developed after the software is well underway and often by groups that are not necessarily part of the original software design team. This does not take advantage of the specific knowledge the original production code developer may have had of the system's intended functionality. Another shortcoming of many test strategies is not testing for defects as soon as possible. The longer in the development cycle a defect has to go to be detected, the higher the cost of fixing that defect.

STRATEGIES AND GUIDELINES FOR MAKING CODE TESTABLE

Testing is an extremely challenging task requiring software design skills significantly exceeding those of ordinary developers.[1] Testing strategies change for different aspects of the development process. For example, an issue rarely addressed during rapid prototyping is testing strategies and the amount of effort required to develop meaningful test suites. To reduce the effort associated with producing test suites and as a means for improving the quality of the test suites, software should be designed with testing in mind from the start. Most Smalltalk development involves the creation of "test harnesses" to exercise code during development. Following a few simple guidelines can help to evolve the normally throw-away test harnesses into valuable permanent tools for improving overall product quality. The following is a synopsis of test strategies and guidelines to be followed during the various phases of Smalltalk-based product development.

Guideline: In object-oriented programming, the class is a natural unit of unit testability. Test suites should be based on verifying the proper functionality of each method (both instance and class methods) associated in with a class.

Guideline: In object-oriented programming, the application or project is the natural unit of integration testability. The integration tests should focus on testing the proper functionality of collections of classes.

Rationale: Integration testing is important because many sources of bugs can be between pieces of code where one method makes assumptions that another doesn't fulfill.

Guideline: Take advantage of the higher productivity of object-oriented development environments, and push more of the unit and integration testing up front into the developers' domain as soon as possible.

Rationale: This approach will permit the developer, who has detailed knowledge of the expected functionality, to develop test strategies more effectively and to begin using them in conjunction with the code development.

Guideline: Validation testing should be performed by an independent testing group in conjunction with the code developer(s), to verify the conformance of the software to the requirements specifications. The validation testing should not only verify proper functionality, but acceptable timing performance also.

Rationale: This approach permits a fresh opportunity to uncover quality problems the original developer may have totally overlooked as well as an independent, and unbiased verification of the actual quality level of the system.

Guideline: Each developer should prepare a formal unit and integration test strategy plan and appropriate test suites for all of the production code and possibly even prototype code. A test plan should be composed of at least two basic components *:

- a brief description of the scope, applicable documents, test strategy
- the actual test procedures, their purposes, test data, and expected results

Guideline: Create a test plan that covers every method in a class and every path within a method.

Rationale: Complete coverage is the objective of the test process. In conventional programming languages, with many lines of code in a module or subroutine, this task can be quite daunting. Fortunately, good OOP style promotes the idea that methods should be short, typically less than ten lines of code. This significantly reduces the complexity of possible paths to be tested within a given method. Another benefit of short methods is that it is much easier to look for bugs in small pieces of code rather than in large program segments.

Guideline: Test cases should be written to uncover general classes of problems rather than one discrete error.

Guideline: Results of testing should typically be logged to a file in a standard format that contains a record of the successful/unsuccessful test of classes and methods that have been performed.

Guideline: If the system permits, the status of the tests should also be displayed directly on the screen using standardized formats. If direct writing on the screen is not possible, then a file should be used.

Rationale: Displaying status to a window during testing is typically problematic as it may cause unwanted cycling of windows, that can complicate testing.

Guideline: Print general messages (i.e., Testing class XYZ) directly on the screen not in windows.

Rationale: The transcript window shouldn't be cluttered with routine and unnecessary status information. If the application has created a bug in the windows there is a possibility that the test routines themselves may be affected.

Guideline: Display errors and log messages along with the date and time in the transcript window.

Rationale: The transcript window provides a window that always exists to record error log messages for later retrieval.

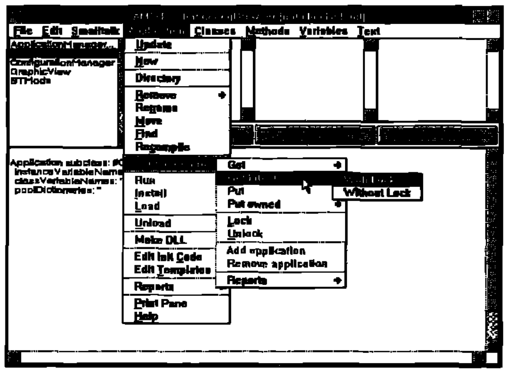
Guideline: Application windows should be at least momentarily displayed for testing.

Rationale: Testing windows in a specific environment can be performed using various third party packages to record keystrokes and mouse movements and then play the sequence back testing for the appropriate response or display in a win-

* The overall high level plan, scope and test strategy are most conveniently documented using a code management system to associate them with an application or project, while the actual test procedures can be associated with the related classes.

ImageSoft

AM/ST™



The original and still premier application manager for Smalltalk/V.™

AM/ST, developed by the SoftPert Systems Division of Coopers & Lybrand, enables the developer to manage large, complex, object-oriented applications. The AM/ST Application Browser provides multiple views of a developer's application.

AM/ST defines Smalltalk/V applications as logical groupings of classes and methods which can be managed in source files independent of the Smalltalk/V image. An application can be locked and modified by one developer, enabling other developers to browse the source code. The source code control system manages multiple revisions easily.

Coopers & Lybrand

Key Features of AM/ST Include:
- Execution timing & profiling tools;
- Cross reference browsing;
- Application hierarchy;
- Automatic documentation;
- Source control;
- Static analysis tools;
- Dynamic analysis tools;
- Your choice of platforms: DOS, Windows, OS/2/PM & Mac.

ChangeBrowser. As an additional tool available for Smalltalk/V PM and Smalltalk/V Windows, ChangeBrowser supports browsing of the Smalltalk/V change log file or any file in Smalltalk/V chunk format.

The addition of AM/ST to the ImageSoft Family of software development tools enhances and solidifies ImageSoft's position as — "The World's Leading Publisher of Object-Oriented Software Development Tools."

Microsoft Windows

1-800/245-8840

ImageSoft™
The World's Leading Publisher of Development Tools

All trademarks are the property of their respective owners. ImageSoft, Inc., 2 Haven Avenue, Port Washington, NY 11050 516/767-2233; Fax 516/767-9067; UUCP address: mcdhup!image!info

dow segment. Unfortunately, this strategy cannot be easily implemented when testing the application for multiple platforms. If the application is to be executed on multiple platforms, a pragmatic strategy for testing the windows is to display all of the windows in sequence. Although this may not be as rigorous as some commercial quality testing criteria, the approach usually has a very high probability of identifying problems that were caused by inadvertent changes to the code somewhere in the system.

Guideline: A library of standard routines should be evolved to assist developers in more easily testing their code functionality. Candidate functions for this library can include routines to force standard keyboard inputs, mouse movements, and possibly even to compare screen displays against previously stored and tested pixel displays. Interactive applications should test themselves (i.e., Dispatchers test models). Some dialects of Smalltalk support an event: or an event:with: message that can be used to easily simulate mouse and keyboard functions for regression testing purposes.

Guideline: During incremental development, unit and integration test suites should be simultaneously developed and exercised.

Rationale: Problems with the code will most likely occur with the new functionality added to the system. The developer can efficiently focus debugging efforts on the small subset of new code rather than a labor intensive debugging of a much larger, more complicated piece of code.

Guideline: Testing strategies need to incorporate mechanisms for testing a system's ability to properly function even when there are significant interrupts to the normal processing flow.

Rationale: Stress testing is one strategy that is intended to capture subtle bugs such as memory leaks or deadlock conditions that may not be evident under other types of testing. Stress testing should include testing over a wide range of voluminous inputs for a long period of time.

Guideline: Testing strategies should include timing as an important characteristic in determining if functionally correct, but nonetheless detrimental, changes have been inadvertently introduced into the system.

Guideline: Do not assume that only the changed class and its subclasses need to be retested. All superclasses as well as subclasses of a new class need to be retested whenever a change is made to a class.[2]

Rationale: One might argue that, theoretically, no testing of superclasses is required if the changes to the class do not access any superclass variables or superclass methods. However, it is impossible to automatically guarantee that such an interaction can not occur. Object-oriented programs must test all of the methods in a hierarchy both above and below the current class position in the hierarchy (subclasses inherit all of the changes in their superclasses, while superclasses can have their apparent functionality overridden by subclasses). There are a number of examples one can conceive where subclasses can change data in their superclasses via global variables or by accesses of superclass instance variables. This situation is further complicated by the possibility that a subclass might inad-

vertently call superclass methods in an incorrect sequence and thereby create an invalid state for other potential classes that need to interact with the superclass. For example, consider a subclass that erroneously instructs a superclass to clear a display window rather than put up a grid whenever a display command is received. Other subsequent operations might be expecting a grid to already be on the screen. Hence, there is no silver bullet in avoiding complete regression testing of class hierarchies whenever a change is made.
Guideline: Do not assume that a superclass test method is adequate for testing a method that overrides it in a subclass.[2] Every method in a class needs to be tested even if it overrides superclass methods that are tested in the superclass.
Rationale: Overriding methods can have significantly different functionality than their superclasses and they need to be tested accordingly. For example, a class may have a method to update a display window and there may be an optimized subclass that has the same overriding method to only update the corrupted part of a display window. Both of these cases need separate test strategies to confirm proper functionality.
Guideline: Every class should have a class method called selfTest that will execute all of the class and instance methods and return a Boolean true if all of the test cases pass or else return a Boolean false if any test fails.
Rationale: The test routines should have standard protocols to leverage off the object-oriented polymorphism and inheri-

EXAMPLE 1:

```
"*************************************
 Project : RegressionTest
 Date    : May 26, 1992
 Time    : 12:01:25

 Introduction
 ============

 Regression testing framework
 partially complete example.

 This application will permit a user to
 include a class method called selfTest
 in each class to test the various
 instance and class methods for proper
 functionality.  If the tests pass, selfTest
 is expected to return a Boolean true, otherwise
 a Boolean false.

 The testing is initiated with

    Object testAll.

 The result of this test is a collection
 of all of the failed classes.

 Dependencies: none

 Invoked By:
 ========

 Object testall

 Description
 ========

 Fully Owned Classes * :

    Object
    .RegTest *
    ..NotOkTest *
    ..OkTest *

 Methods of Partially Owned Classes: #
    #testAll defined in Object class.
 *************************************"!

 Smalltalk at: #Object ifAbsent: [
 nil subclass: #Object
  instanceVariableNames: ''
  classVariableNames:
   'RecursionInError Dependents RecursiveSet '
  poolDictionaries: '']!

Object subclass: #RegTest
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''!

RegTest subclass: #NotOkTest
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''!

RegTest subclass: #OkTest
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''!

!NotOkTest class methods !
selfTest
  "Public-"
   ^false! !

!RegTest class methods !
selfTest
  "Public-"
   ^true! !

!Object class methods !
testAll
      "Public-Find all of the classes which implement the class
       method selfTest and execute the method for those classes
       returning an ordered collection of all of the failed classes.  To
       test, execute

          Object testAll

      with a show it."

   | allObjectSubclasses testClasses okClasses failedClasses |
   testClasses := OrderedCollection new: 30.
   okClasses := OrderedCollection new: 30.
   failedClasses := OrderedCollection new: 30.

   self withAllSubclasses
      do: [:eachClass |
            (eachClass class includesSelector: #selfTest)
               ifTrue: [testClasses add: eachClass]].

   testClasses
      do: [:each |
            each selfTest
               ifTrue: [okClasses add: each]
               ifFalse:[failedClasses add: each]].
   ^failedClasses! !
```

tance that promote increased productivity and reusability. It is important that this protocol is followed as a standard so that testing can be automatically accomplished even when applications, projects, or frameworks from different developers are combined into larger systems. By following this guideline's standard protocol, any or all of the code in the system can be regression tested for any unexpected side effects using a simple method such as

```
Object testAll.
```

where the method selfTest would be included in each class to be tested (see Example 1).

Guideline: Every class should have an example or set of examples showing its use as shown in Example 1.

Guideline: Every example should have an executable test case in a comment as shown in Example 1.

AN ERROR CHECKLIST

A number of common errors can be the basis for testing routines.[1] These tests ** can include checks for:

1. Input and output validity
2. Missing inputs
3. Range violations
4. Proper handling of incorrect inputs
5. Unwanted interactions with other methods and classes via global or class variables or pool dictionaries
6. Integrity of data structures (e.g., do persistent objects match the current classes' instance variables?)
7. Proper error handling (e.g., can a recursive error cause the system to abort?)
8. Nonexecutable code
9. Infinite loops
10. Testing at the limits of data structures or around minimum and maximum values

CAVEATS

Guideline: Immediately seek help if an impasse is reached during testing and debugging.

Rationale: Days and even weeks of effort can be wasted on glaring errors invisible to one person but readily apparent to another.

Guideline: Don't get too caught up with regression testing of code!

Rationale: The downside of depending too much upon regression testing of code for quality assurance is that the developers can place disproportionate effort in regression testing while abandoning common-sense practices in their code development and design review practices.

Guideline: Budget at least one-third of the total development effort for testing.

Rationale: Irrespective of whether O-O or conventional languages are used, test code is a significant percentage of the total product effort and in some cases can contain more code than the actual runtime functionality.[3] The test suites are typically of the same magnitude of source lines of code as the product development effort. On the basis of conventional practices, this activity can be expected to account for at least one third of the programmers' total effort. Furthermore, testing of mission critical applications where human life is at risk can consume over 80% of the total development effort.

CONCLUSION

Smalltalk offers significant potential for overcoming many reliability shortcomings of other programming systems, but extensive reuse of Smalltalk-based libraries requires some consistent standardized protocols and testing approaches. Unfortunately, software testing and quality assurance issues are often unnecessarily overlooked in the normal rapid prototyping developments that are so well suited to Smalltalk. This article has highlighted testing guidelines and proposed an initial standard framework for testing of reusable code modules. The guidelines, if followed from the initial phases of development, can result in more robust, testable, and hence reusable frameworks with a minimum of additional effort. ■

ACKNOWLEDGMENTS

The insightful reviews of this document by Professor Ralph Johnson, University of Illinois at Urbana and Ms. Kate Funkner of the Allen-Bradley Software Quality Assurance Group are gratefully acknowledged.

REFERENCES

1. Myers, G. THE ART OF SOFTWARE TESTING, Wiley Interscience, New York, 1979.
2. Perry, D.E. , and G.E. Kaiser. Adequate testing and object-oriented programming. JOURNAL OF OBJECT-ORIENTED PROGRAMMING, January/February, 1990.
3. Rettig, M. Testing made palatable, COMMUNICATIONS OF THE ACM, 34(5): 25–29, 1991.

** Since unexpected inputs are a major source of software failures, the importance of these tests can not be overemphasized.

Ed Klimas is Managing Director of Linea Engineering Inc., a supplier of custom object-oriented based solutions for automation and industrial applications. Ed, along with Dave Thomas and Suzanne Skublics of Object Technology International, are coauthors of an upcoming Addison-Wesley book titled SMALLTALK WITH STYLE *that covers*

...continued from page 1

tions, a second may be working on database aspects, a third may be working on the user interface, and yet another person may be working on error management and exception handling. In programming terms, we then create separate components in our development environment. In ENVY, the term used is *application*. An application represents a collection of classes that together serve some useful purpose. A complete system would then be constructed from a number of such applications, each of which represents a standalone piece of functionality. We use the term *component* as an umbrella concept that covers classes, sets of related classes (applications), and sets of related applications (configurations).

“The critiquing process is very similar to the original coding process; it has the same look and feel.”

THE MECHANICS

The mechanics of the review process consists of at least three steps:

1. Preparing the components for review
2. Actually reviewing the code
3. Integrating the review results into the next version of the component

The review process begins when I, as the author of a component, assess it fit for a peer review. This happens when I have implemented many of the classes corresponding to the objects that I have discovered through the initial analysis and design phases. At this point, all the details may not have been completely fleshed out. I version all of the classes in each of my applications with a distinguishing version label: [For Review, Mar 22]. During development, we have adopted the convention that we version all our components using a stylized date stamp: [Mar 22], [May 02, 7 PM], etc. This is intuitively more meaningful than version labels such as 0.99, 0.995 etc. Of course, when we make our external release for customers, the labeling follows more conventional guidelines.

Having labeled all of the classes with the [For Review] timestamp, I then label the version of the applications containing the classes with the same label as that of the classes. It is important to note that under ENVY, even methods added to classes that are defined outside of my component (e.g., `Stream`, `String`, `Collection`) are maintained in the context of the application. Thus, these class extensions also are susceptible to the same versioning rituals and are also subject to peer review. I now collect all the [For Review] applications and build a configuration consisting of these applications. ENVY employs the notion of a configuration map to group related sets of compatible applications. You must specify the particular versions of the applications that are released into the configuration map. Accordingly, I now construct a configuration map, called Credentialing System, for example, that consists of the [For Review] versions of the associated applications. I now version the configuration map itself with the same [For Review] label.

At this point, I inform a technical peer, say Joe, that he should review all the software contained in the Credentialing System configuration map. The specific version of the configuration that he should review is the one labeled [For Review, Mar 22]. Depending upon the bulk of the code to be reviewed, it may take Joe anywhere from two days to a week to review it. The amount of code to be reviewed should be of manageable size; the reviewer should be able to do the job in two or three days.

After I have submitted all my code for review, I can continue developing my components without waiting for the review results. Basically, I start a branch from the [For Review] version, and continue on a stream of development. Joe starts from the same version and opens up a parallel stream of development where he'll carry out the review activities. What are these review activities? This is the topic of the next section.

THE CRITIQUE AND REVIEW PROCESS

Following our example, the reviewer loads the Credentialing System [For Review, Mar 22] configuration into his Smalltalk image. He creates a new edition or working copy of all the applications in the configuration. For the reviewer to do an adequate job, there need to be some commonly accepted guidelines for critique and review. We will discuss guidelines in a subsequent section. The critiquing process is very similar to the original coding process; it has the same look and feel. You are adding comments to and changing, improving, deleting, reformatting code, etc. in the classes using the same browsers and tools the original authors use. The significant thing to note here is that the reviewer is in place adding review comments in the body of the method or even fixing the algorithm or the control structures within a method. The reviewer can factor code better; he can add new methods that may serve as code factors. He can delete methods that he considers obsolete or otherwise unnecessary. He may add review comments in the body of the method, just like he would add a regular method comment. He can reformat the method or introduce better indentation to make the code appear more perspicuous.

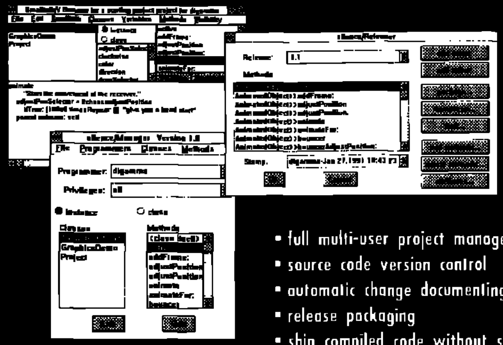
All these give rise to a nice asynchronous review process that doesn't impact the stream of onward development by the original author. The reviewing activity seems very much like coding, except that you are now largely reading someone else's body of code and applying your critiquing skills to it. The desired result is for the technical team as a whole to produce deluxe, sterling code. It is incumbent on the reviewer to bring to bear on someone else's code all the good programming practices he has learned over the years. The idioms he has learned to express a certain piece of computation may be far more elegant and efficient than ad hoc "hammer and tongs" spaghetti code. This is particularly a problem with new Smalltalkers schooled in procedural thinking and those unfamiliar with the highways and byways of the class libraries. The reviewer is able to transfer Smalltalk programming nuances hitherto unknown to the author of the code.

“When instituting a code review process, the organization must a priori agree upon the guidelines with respect to which the review is to be conducted.”

When the reviewer finishes going through all the classes in each of the submitted applications, he versions all the classes he has touched with a new distinguishing version label: [Reviewed, Apr 10]. In turn, he versions all the affected applications with the same version label. Finally, he versions the submitted configuration map (Credentialing System, in our example) with the [Reviewed] label. The original author is informed that his code has been reviewed and the results of the review are in Credentialing System [Reviewed, Apr 10].

s·i·l·e·n·c·e

a collection of tools for project management and code delivery



- full multi-user project management
- source code version control
- automatic change documenting
- release packaging
- ship compiled code without source
- reconfigurable installation tool
- change log browser and restorer
- code performance profiling

$149.95

NEW! *UFO* persistent object toolkit
File any object in or out, flatten any object to a null terminated string
Available for the introductory price of **$49.95**
Included free with every **silence** purchase!

digamma solutions

Unit 6, 387 Spadina Avenue, Toronto, Ontario, Canada, M5T 2G6
Phone: (416) 351-8833 Fax: (416) 408-2850

[illegible]

THE INTEGRATION PROCESS

When the reviews come back (in the form of newly versioned components), the author uses a variety of differencing tools to quickly pinpoint the areas of code that have been critiqued. If he agrees with the proposed changes, he can fold them into the next version of the component. This is easily done with environments such as ENVY that provide a rich set of tools that can be used to quickly browse through the differences between any two versions of a given component. The author, for example, would do a Browse Differences between the [For Review, Mar 22] version of his application and the [Reviewed, Apr 10] version. The differencing tools pinpoint the precise differences between the original method and the annotated method.

To start off the integration process, he opens a new edition or working copy of his application that was originally versioned [For Review, Mar 22]. He then goes about systematically incorporating the changes suggested by the reviewer. In some cases, this may involve a merging of the two versions of a method. If the reviewer has produced a new version of a method that the author deems superior to his original version, he folds the new version wholesale into his working copy of the class. In many cases, the reviewer may simply have made some suggestions or requested the author to clean up the code in some way. It is then up to the author to respond to these suggestions appropriately. Quite often, the reviewer may have misunderstood the intent of the particular technique the author has employed. In such cases, the author may choose to ignore the reviewer's suggestions. It is also possible that the fix the reviewer suggested would have an adverse impact on a lot of the client code that the reviewer is not aware of.

To close the loop, the author, after systematically going through the reviewer's feedback using differencing browsers, produces a new version of the component. It is quite possible that the author has added new code to the [For Review] versions of the components since he submitted the code and received the reviewer's versions. Now that he has finished integrating the reviewer's comments, it is an opportune time to integrate the code developed since the time of submission. Sophisticated programmers may choose ever so slightly different ways to integrate the new code. Thus proceeds the evolution of the software components—an evolution enriched by the collective insights of the author's peers.

Ultimately, the author can submit his increasingly polished software to a different set of reviewers. The new reviewers will have the benefit of the first reviewer's insights and do not have to hoe the same row. They may shed completely different perspectives on the submitted software. This can only improve the quality and maintainability of the software. In addition, the code repository now contains a complete audit trail of the evolution of a software component, including all its review versions.

REVIEW GUIDELINES

When instituting a code review process, the organization must a priori agree upon the guidelines with respect to which the review is to be conducted. Failure to do so causes tension and misunderstanding among team members. A big part of the code review pertains to coding style. This has been amply covered in the Smalltalk with Style columns in previous issues of THE SMALLTALK REPORT.[1, 2] The Smalltalk style issues include proper code indentation, choosing proper variable names, good method comments, and so on. We enumerate some of the empirical guidelines that we employ in the review process below.

BASIC GUIDELINES

What to watch out for in a code review:

- Poor indentation. This is the culprit in a lot of sloppy code. Proper indentation makes the code clearer and a pleasure to read. Indentation also gives rise to heated disagreements among team members. "Hey, that is my personal writing style. Don't mess with it." Good, reusable classes have lots of readers. Think about the users who will have to read your code long after you have left the scene. A little uniformity and consistency in style goes a long way. The Smalltalk with Style guidelines are certainly a good place to start. Writers and editors often adhere to the guidelines in THE CHICAGO MANUAL OF STYLE or THE ELEMENTS OF STYLE by Strunk and White for similar reasons.

- No class comments. Standard Smalltalk/V does not have tool support to embed class comments. Several commercially available development environments, however, do have explicit tool support to add class comments. Use them. A good class comment should pithily describe its purpose and how it is intended to be used. Additionally, it can also include some global implementation notes, a description of its instance variables, and any special algorithm used in implementing the class.

- No method comments. It is frustrating to figure out the intent of a long rambling method if it doesn't have accurate comments. Even well-written methods can use comments to explain operating conditions, and pre- and post-assertions. Implementation notes on why a particular data structure has been chosen can also be helpful.

- Gratuitous comments. This is the other extreme. Code that is extremely clear and self-documenting doesn't need comments that add no value:

```
foo: anInteger
    "Set foo to anInteger."
    foo := anInteger

isEditable
    "Return false."
    ^false
```

- Misleading comments. Code doesn't reflect comment or vice versa. Often, comments refer to method parameters that have long since been replaced by something else, but the comments themselves haven't been updated. The other common infraction is that the comment claims to do one thing, while the code does something else:

```
add: anAssoc
    "Answer anObject. Add anObject to the receiver
     if the receiver does not already contain it."
    "Save in the trades table"
    | ok |
    CursorManager execute change.
    trader query: (self update:anAssoc).
    ok := trader fetchResults.
    CursorManager normal change.
    "save locally"
    (retrievedAllFlag | cacheFlag)
        ifTrue: [super add:anAssoc].
    ^ok
```

Bad comment—it refers to an obsolete argument and claims to return one thing, but actually returns something else.

- Cryptic comments:

```
link: aLink col: colNbr
    "Return column data"
```

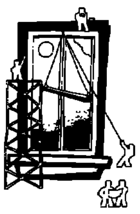
- Annoyingly informal comments. "Can you believe XYZ Co. did this? Sheesh!" "Joe really messed this up," "If the poor sap gets to this method, he's really asking for it." Such comments may be okay and may even promote camaradarie if the software is going to be used only within your limited workgroup, but for production quality software that will be used by anonymous customers, it represents poor taste.

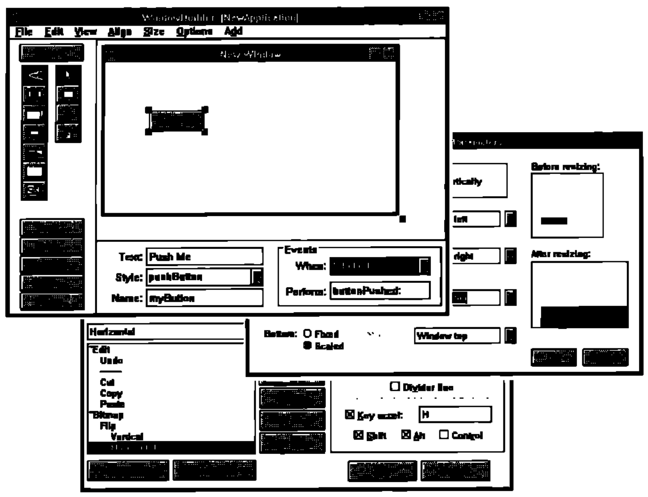
- Bad argument names. Beginning Smalltalkers, particularly C programmers of the argc-argv school, commit this violation frequently. Using nonintuitive argument and variable names like arg, aParm, parm1, parm2, tmp1, tmp2, or even anObject doesn't serve anyone. It is a nightmare to figure out the classes of the objects that are participating in the computation. Likewise, method selector names shouldn't be misleading; they should be intention revealing.

COOPER + PETERS

WINDOWBUILDER

The Interface Builder for Smalltalk/V





The key to a good application is its user interface, and the key to good interfaces is a powerful user interface development tool.

For Smalltalk, that tool is WindowBuilder.

Instead of tediously hand coding window definitions and rummaging through manuals, you'll simply "draw" your windows, and WindowBuilder will generate the code for you. Don't worry — you won't be locked into that first, inevitably less-than-perfect design; WindowBuilder allows you to revise your windows incrementally. Nor will you be forced to learn a new paradigm; WindowBuilder generates standard Smalltalk code, and fits as seamlessly into the Smalltalk environment as the class hierarchy browser or the debugger.

Our new WindowBuilder/V Windows 2.0 is now available for $149.95, and WindowBuilder/V PM is $295. Both products include Cooper & Peters' unconditional 60 day guarantee.

"... this is a potent rapid application development tool which should be included in any Smalltalk/V developer's environment."
- Jim Salmons, The Smalltalk Report, September 1991

For a free brochure, call us at (415) 855-9036, or send us a fax at (415) 855-9856. You'll be glad you did!

COOPER & PETERS, INC. (FORMERLY ACUMEN SOFTWARE) 2600 EL CAMINO REAL, SUITE 609 PALO ALTO, CALIFORNIA 94306 PHONE 415 855 9036 FAX 415 855 9856 COMPUSERVE 71571,407

- Coding anachronisms. This is a case of old habits dying hard. New Smalltalk releases and development environments often introduce new programming idioms. Always accessing classes through symbols—e.g., **(Smalltalk at: #HeatingUnit) raiseTemperature**—is no longer necessary if the development environment allows you to explicitly specify class prerequisites. Furthermore, cross-referencing facilities such as browse methods that reference the class HeatingUnit will not show up this method since the class object is buried inside a defensive expression. If I were to rename the class HeatingUnit, I might miss making a change to the method, and that will cause a runtime error in the future. Referring to classes indirectly through class names goes against the grain of classes as first-class objects. Of course, there may be exceptional conditions under which this may be warranted.
- Using needless temporaries. Beginning Smalltalkers tend to gratuitously use temporary variables to do iteration. There are a phalanx of iteration idioms like collect:, select:, inject:into: that do the job more efficiently and make the intent of the code clearer.
- Needless multiple exits from a method. Another defensive programming malady that could be remedied by proper use of the conditional control constructs and good indentation.
- Poor distribution of responsibilities. This is really poor design that percolates into the coding process. This manifests itself in heavyweight classes with 100 methods that do a lot of things. Often, you expect a certain responsibility to be discharged by a certain object; instead, in code it manifests itself as a complicated multi-keyword method that requires a lot of hand assembly to use. Reviewers can suggest, in such cases, how these methods can be rewritten, refactored, etc.
- Defensively lodging "global" behavior in Object. If a certain behavior is expected to be global within your particular domain, find an appropriate class in your domain to lodge this behavior. If every class in your domain must have this behavior, create an abstract superclass that contains this method and have every one of your classes inherit it. Alternately, you can lodge this behavior in a class method of an appropriate class in your domain. Blindly putting the behavior in Object causes all classes in the system to be susceptible to any side effects that it may cause. Reviewers may suggest alternatives appropriate to the context in which the code is meant to be used.
- Put global class methods in Behavior, not Object. I have come across several Smalltalkers who put catch-all class behavior in Object. This is too defensive. Behavior is the root superclass in which you should lodge behavior that is expected to be inherited by every class in the system.
- Unnecessarily using isKindOf:, isMemberOf:. This is not considered good programming style. You are referring to a class directly via a hard constant. This is not only inefficient, but also less reusable. Instead, use polymorphic test mes–

sages: isString, isNurse etc. In this case, you'd implement isString methods in Object and String, returning false and true, respectively. If you are sure that the receiver of the isKindOf: method is going to be an object of a class that is directly or indirectly inherited by a particular superclass, then put the false test method in that superclass. For example, if you are sure the receiver is always going to be some kind of a subpane, put the false test in SubPane, not Object.

“As far as possible, code should be optimized for the environments that it is supposed to work on; platform differences should be dealt with by partitioning code into platform independent and platform specific portions.”

- Archaic, old, effete code should be weeded out. Code should keep step with emerging idioms, and newer and better features in the underlying class libraries. For example, if a new version of your base environment provides richer protocol to insert an association in a dictionary use that. It would make your code more compact. For example, ENVY has a Dictionary method at:ifAbsentPut:, so code that is originally written as:

```
myConstants := Smalltalk
    at: #MyConstants
    ifAbsent: [Smalltalk at: #MyConstants
                        put: #(a b c)]
```

  can have a more modern rendition

```
myConstants := Smalltalk
    at: #MyConstants ifAbsentPut: [#(a b c)].
```

  A common counter argument to this guideline is, “Hey, I have to make my code work under multiple environments.” This means programming to the least common denominator of available idioms. As far as possible, code should be optimized for the environments that it is supposed to work on; platform differences should be dealt with by partitioning code into platform independent and platform specific portions. ENVY, for example, provides explicit tool support for cross-platform development.

- Writing “to be implemented by subclass” in the comment and then not expressing that intent in code. If a method must be implemented in each and every subclass, then that intention must be expressed via appropriate programming idioms. Smalltalk/V has implementedBySubclass and Objectworks for Smalltalk has subclassResponsibility methods for such purposes. Putting self halt is not acceptable. Similarly for subtractive inheritance, Objectworks provides a shouldNotImplement idiom.

- Putting self halts to trap exception condition is not acceptable. While this condition may not manifest in the course of normal developer work, the moment it hits the customer's hands, it is sure to trigger the exception and cause an unpleasant walkback window to pop up.

- Using inappropriate data structures. Programmers often use familiar data structures because it gets the job done. A very common example of this is the use of the class OrderedCollection. OrderedCollection is very malleable and very flexible. It responds to a wide variety of protocol and is a friend of the defensive programmer. However, all this comes at a price: space and time overhead. In cases where it is clear that a fixed size collection can do the job, Arrays should be used. In cases where elements in the collection are retrieved in batches, a LinkedList is perhaps a more appropriate data structure. Reviewers should be on the lookout for this.

- Defensive parentheses that do not contribute to code clarity:

```
(aValue isNullValue)
    ifTrue:[^aValue].
(aValue isNullValue or: [(aValue isKindOf: DateTime)])
    ifTrue:[^aValue].
```

  A hopefully better rendition is:

```
^(aValue isNullValue or: [aValue isDateTime])
    ifTrue: [aValue]
```

- Use CharacterConstants or messages instead of literals to represent characters. This makes it more readable. $ is less readable than Space or Character space.

- Use Stream protocols instead of lengthy concatenation of strings.

- Use nonevaluating conjunctions or disjunctions for some ifTrue:ifFalse: blocks that return booleans:

```
Bad code:
    aBool ifTrue: [^5] ifFalse: [^false].
Good code:
    ^aBool and: [5]
```

ADVANCED GUIDELINES

- Inefficient algorithms. Reviewers should watch out for nonpolynomial time algorithms and should suggest faster algorithms if one exists as opposed to, say, O(n^3) algorithms.

- Poor factoring of code.

- Poor construction of inheritance hierarchies.

- Poor separation between platform-specific and platform-independent code. It is poor practice to partitioning platform-specific code across classes (a programming language concept) as opposed to across applications (a programming environment concept).
- (Im)proper partitioning of methods into public and private methods. For example, initialize and release methods should be private.
- Proper initialization and destruction of data structures (initialize and release).

"Just as a good general makes the soldiers around him better, so, too, should the benefits of having a good programmer accrue to other less experienced programmers on the team."

- Messages sent but not received (implemented). This is a potential runtime error.
- Messages implemented but not sent. This is quite possible, if the component is intended to be used as an extensible framework. However, if you are delivering a standalone application, these methods are candidates for elimination. You should make sure they are not called via perform:.

SOCIOLOGICAL EFFECTS

The impact of the code review process as described in this article is most effective in organizations that actively encourage programmers to critique each other's work. On the other hand, in workgroups that have a "cream puffs, marshmallow" culture, there is probably a fine line, the crossing of which may yield undesirable sociological results that counter the benefits of a stringent code review. Recently, while drawing up a blueprint of Smalltalk software development for his company, an MIS manager commented, "Of course, we are going to write good code. Peer fear will ensure that nobody writes sloppy code." Organizations and workgroups that can distinguish between an honest critique and a personal barb are most likely to benefit from a rigorous review process. It is also likely that some developers will be more reluctant than others to submit their code for review. This happens for a variety of reasons, not the least of which is a fear of being unfavorably critiqued. Just as a good general makes the soldiers around him better, so, too, should the benefits of having a good programmer accrue to other less experienced programmers on the team.

Universal Database
OBJECT BRIDGE ™
This developer's tool allows Smalltalk to read and write to: ORACLE, INGRES, SYBASE, SQL/DS, DB2, RDB, RDBCDD, dBASEIII, Lotus, and Excel.
Arbor Intelligent Systems, Inc.
506 N. State Street, Ann Arbor, MI 48104 (313) 996-4238 (313) 996-4241 fax

Code reviews should be viewed as a rigorous software engineering activity, not a mindless chore undertaken half-heartedly to adhere to some "feel good" corporate software development guidelines. Good Smalltalk programmers tend to read a lot more than they write. They tend to adopt the idioms of other good programmers while inventing a few of their own. Ward Cunningham once observed, "Good class libraries whisper the design in your ear." While browsing through other people's code, I often say to myself, "That's a great idea! why didn't I think of that?" Kent Beck has termed these voyages of discovery the "Aha! experience." [3]

Well-written code is a pleasure to read, and it becomes quickly apparent if a class is reusable or not. People try to imitate and emulate well-written code. Well-organized code reviews have the benefit of bringing to bear every team member's perspectives and experiences to the total quality of the software being shipped. Even with a cohesive review process, there are plenty of ways in which team members can express their own unique styles in the delivered code. Good software engineering practices as expressed in Smalltalk code can only propagate the positive effects and thus result in more gratifying Aha! experiences for everyone. ■

REFERENCES

1. Klimas, E., and S. Skublics. A matter of style, THE SMALLTALK REPORT 1(2):1–5, 1991
2. Klimas, E., and S. Skublics. Tips for improved Smalltalk reuse and reliability, THE SMALLTALK REPORT 1(6):11–14, 1992.
3. Beck, K. Essays and A-Has, HOOPLA! January, 1988.

S. Sridhar is a senior member of technical staff at Knowledge Systems Corporation in Cary, NC, where he is actively applying Smalltalk to a variety of software engineering problems. He came to KSC from Mentor Graphics Corp. where he was the project lead for Mentor's second generation design management environment developed in C++. Prior to that he worked at Tektronix, Inc. for four years on Common Lisp and Smalltalk/80 product development. He can be reached at 919.481.4000.

Alan Knight

What's wrong with OOP?

Things object oriented have been receiving a lot of attention lately. Some is justified interest in an important emerging technology, but a lot of it is just hype. It is apparent that in some circles, "object-oriented" has become the buzzword of choice.

Everywhere you turn things are described as *object oriented*. Operating systems, windowing environments, and programs of every description are now labeled O-O. There's more restraint with programming languages, perhaps because there's a clear definition of what an object-oriented language is. For programs, it's only necessary that the designers think about the problem in an object-oriented way, or that something in the program be called an object.

At the same time, object-oriented programming, design, analysis, databases, graphics, and other functions are prescribed for all problems. There is a silver bullet, we've got it, and for only a few dollars you can have your own compiler! Triple your productivity overnight! Plus, if you act now, we'll turn software into a rigorous engineering discipline at no extra charge! Even the editors of BYTE magazine, who really should know better, write about how O-O promises to make computer programming accessible to the right-hemisphere, creative people who have traditionally been excluded. I could go on about that one for quite a while, but I'll restrain myself.

This much hype inevitably provokes a backlash. Those who have already embraced some other method of saving the world resent OOP's popularity and look for flaws. Others without detailed criticisms nevertheless remain skeptical of the inflated claims and suspect hidden drawbacks. The rest of this column may remind you of a few arguments you've been in, as I'll be going through some representative questions and criticisms, with comments and some of the replies. If you haven't yet had to deal with these, perhaps you'll be better prepared when the time comes.

WHAT'S THE CATCH?

Devon T. Caines (caines@andrews.edu) writes:

> **Just one quickie question.**
>
> **What are the real disadvantages of OOP? There must be a down side. True, the learning curve is one, but I don't consider that serious enough.**

This is a typical "OOP novice" question. Surely people wouldn't devote so much effort to selling you something that didn't have drawbacks. Is the learning curve too steep? I've heard it's inefficient. Are the claims of easier maintenance and reuse borne out in practice? If OOP encourages reuse, how come there isn't more of a software components industry? Are there real companies doing major projects with OOP? What kind of success stories or disasters have there been?

Let's start with the disasters. One well-publicized failure occurred at Cognos, an Ottawa company that makes 4GL tools. A few years ago they tried to switch from C to Eiffel as the implementation language for their main product. The project ran into severe difficulties and was eventually abandoned. Burton Leathers is a Cognos representative who has spoken and written about the project in several places, including an article in the July 1990 HOOT (HOTLINE ON OBJECT-ORIENTED TECHNOLOGY). The problems with the project are said to include immaturity of the Eiffel tools at the time it was undertaken, the attempt to move wholesale into OOP without adequate preparation, and general bad management. The order of importance of these factors depends on who you talk to.

On the other side, there are a lot of OOP success stories. A particularly impressive one involving Smalltalk was recently described on the net by Bruce Samuelson (bruce@utafll.uta.edu), who credited the May 11th COMPUTERWORLD magazine as the source. A company named EDS (Electronic Data Systems) recently did a test project, rewriting a system using PL/1 and a relational database into Smalltalk with an O-O database. The PL/1 system was quite new, so specification, design, and test documents were still available, as were the original implementors. Using a team experienced with Smalltalk, they achieved incredible productivity gains on the order of 14:1. I haven't seen this confirmed, so if you want to use it as an example, it would be best to check the details.

MAINTENANCE AND REUSE

One of the principal claims of OOP is that it allows easier maintenance and code reuse. These are not easy things to measure, and OOP detractors often argue that these gains are imaginary.

For example, Ravi Kalakota (kalakota@ut-emx.uucp) wrote:

The oft-cited advantages of the O-O approach have been code (and design) reuse, easy extensibility and modifiability. O-O has brought about substantial reduction in the size of new code that has to be generated and also the size of the overall system. As the code size becomes more "concentrated," the complexity of the system increases dramatically (at least in my opinion). Does this complexity affect maintenance?

Probably not in a simple project. But in a large system....

Perhaps because these things are so hard to measure, they attract a lot of interest on both sides. There were a lot of anecdotal stories posted, as well as references to several studies purporting to prove either that O-O improved maintenance or that it made it more difficult. Many disagreed with the basic idea of this post that complexity necessarily increases as code size is reduced. The general perception is that in normal cases reduced code size reduces complexity. In APL, for example, part of the reduction in code size is due to the use of one-character identifiers, which certainly reduces readability. Even there, though, most of the reduction is due to the language providing very powerful operations.

There was a strong consensus among all parties on the idea that maintainability is not an inherent property of any programming language or paradigm. There have been bad programs written in O-O languages, and will undoubtedly be many more in the future. Designing for maintainability is the essential part of building a maintainable system. Whether O-O methods really assist in this or not remains in dispute, and likely will until many more studies have been done.

As Ralph Johnson (johnson@cs.uiuc.edu) writes:

You can build poorly structured systems in any language. The question is whether you are able to build well-structured systems.

I build well-structured systems in Smalltalk that are relatively easy to understand. My feeling is that OOP makes it easier to build well-structured systems. It is easier to see the design, so there is more motivation to do it right.

OOP IS UNNECESSARY

Then there are those who actively try to find fault with particular aspects of OOP. Some claim that most of OOP is not really new, and that the new parts are not important. In this view, the productivity gains claimed for OOP are primarily a result of its use of standard ideas on abstract data types and information hiding. Inheritance and dynamic binding are considered to be either insignificant or actively harmful.

This type of argument often occurs in a discussion of the relative merits of Ada and C++. This is one argument where I don't have a strong opinion. I'd hate to have to work in a language without the O-O features I'm accustomed to, even if it did embody good software engineering principles. On the other hand, a language based on C, with all of its "features," doesn't appeal to me either. Call me spoiled, but I like arrays that know their own size, and I like languages that at least allow for the possibility of runtime checks. I know that in C++ it's possible to write or get hold of array classes that work properly, but the default is still C, and a lot of code uses the default.

"Call me spoiled, but I like arrays that know their own size, and I like languages that at least allow for the possibliity of runtime checks"

Of course, there are people who like C, and many of them really like it and don't see why you would ever want another language, even one as close as C++. Martin A Leisner (leisner.henr801c@xerox.com) writes:

OOP is a buzzword for "good design practices".... I've been doing OOP in C for years (C++ just enforces more a disciplined programmer doesn't need). If OOP helps make good designs, great. But good designers make good designs, not the language/system.

I have to agree that good designers are important, perhaps even more important than good languages. However, from my own experiences in having to do some work in C for the past couple of months, I (speaking as a good designer) am getting extremely frustrated with the lack of both OOP and software engineering facilities. One way to make C++ look good is by writing in plain C. The inability to generalize operations to work on different types, the need for function pointers everywhere, and the lack of enforcement on information hiding all make it very difficult. Ralph Johnson (johnson@cs.uiuc.edu) had a particularly good line for this:

Information hiding can be implemented by self-restraint. You can tell yourself that clients should never use a particular fact about a component and then believe yourself when you design the components. Self-restraint works almost as well for implementing information hiding as it does for birth control.

C is a pretty easy target, so it's not really fair to take that as our representative non-O-O language. Suppose we use Ada instead, which does have support for information hiding, genericity, and many other nice features. Does the full power of O-O really gain us much over this?

Peter Hermann (ph@rus.uni-stuttgart.dbp.de), who apparently doesn't think so, writes:

In an excellent paper "Object Oriented Extensions to Ada: A Position Paper for O-O Ada Panel" TRI-Ada '90 p.92-94 ISBN 0-89791-409-0 Schwartz, Jack H. wrote: Conclusion: The current definition of the Ada language already includes the best features of object-oriented languages, namely:

- **modularity**
- **encapsulation**
- **separate compilation**
- **genericity**
- **dynamic instantiation**
- **lexical overloading**
- **exception handling**

and deliberately excludes many questionable and dangerous features such as

- **inheritance**
- **dynamic binding**

for the very reason that they compromise the requirements of large scale software engineering. It is, therefore, inadvisable to attempt to extend the Ada language to include features that conflict with Ada's primary goals: maintenance, reuse, and programming-in-the-large.

“Certainly inheritance can be badly misused by the inexperienced or by the experienced in a hurry. Does that mean it should be left out of a language?”

This paper, oddly enough, seems to consider things like separate compilation and exception handling to be features of object-oriented languages. I'm not sure if that was an attempt to minimize the importance of inheritance and dynamic binding by placing them amid a long list of features or if it's due to just listing all the features in C++ and attributing them to OOP. That aside, this post does raise the interesting question of how important inheritance and dynamic binding are to OOP, and to good software engineering in general.

INHERITANCE AND DYNAMIC BINDING CONSIDERED HARMFUL

Inheritance is a very powerful and useful feature, especially in a prototyping environment. Is it really "questionable and dangerous" as well? This has been discussed in the O-O community for some time, and there seem to be two main issues. The first is inheritance for subtyping vs. inheritance for code reuse. On this, Ralph Johnson (who I keep quoting because he keeps writing good stuff) writes:

I see the use of inheritance as one of the differences between people interested in developing reusable software and those interested in getting applications out the door.

People who are good at developing reusable software want to develop elegant software and dislike using inheritance just for code reuse. However, people who want to deliver applications as soon as possible think that overriding arbitrary methods is the greatest thing since text editors. Both are right. They have different goals, and thus different criteria for evaluating programs.

Inheriting for code reuse is fine in a prototyping environment, but those sorts of relationships shouldn't normally make it into production code. The standard Smalltalk example of this sort of inheritance is Dictionary being a subclass of Set, even though the interfaces are quite different. This relation is now enshrined in tradition, but to my mind it would be much more sensible for both to be subclasses of collection, implemented in terms of a third class HashTable. This would achieve roughly the same level of code reuse, make the operations of Set and Dictionary trivial, and make it much easier to write sets or dictionaries with alternative implementations.

One reason this sort of thing doesn't get done is simply laziness on the part of the programmers. It's easier to inherit and change the interface than to write a new class, and there's a reluctance to write classes that do little more than translate their operations directly into an underlying representation. Programmers will use an OrderedCollection to implement a stack rather than writing a Stack class, even though OrderedCollections have many operations inapplicable to stacks. One of these operations applied anywhere in the code could lead to a very hard-to-find error.

The second issue is that of composition vs. inheritance, which is closely related. If you shouldn't inherit for code reuse, then to reuse code you have to build new classes that encapsulate the code you are trying to reuse. This can be done by writing a class around an ADT as in a stack implemented by an OrderedCollection. It can be done by writing a class that includes instances of several other classes and manages the relationship between them (I'd say this was the normal case in OOP development). Or it can be done by writing classes that manage somewhat arbitrary collections of components (such as a windowing class that manages a group of sub-windows).

Richard Thomas (thomas@qut.edu.au) writes:

Inheritance is the biggest problem and danger of OOP....Most students have problems deciding when to inherit and when to import. Often they will inherit too

much and generate a messy fettuccine (a la spaghetti) class structure. Unfortunately this isn't limited to student efforts. Many examples of OOP in the literature and in class libraries use inheritance just about every time they want to access a feature. This is wrong! Overuse of inheritance leads to a very complex, nonintuitive class hierarchy that takes a considerable amount of effort to understand....Inheritance for code reuse (not subtyping) allows you to defeat the principles of information hiding. Either I view a subclass on its own and have important features that this class depends on hidden from me in superclasses. Or I look at the expansion of a subclass to include its superclasses and have to interpret the entire mess and figure out all the dependencies.

So it appears there is a valid point in the idea that inheritance is "dangerous." Certainly inheritance can be badly misused by the inexperienced, or by the experienced in a hurry. Does that mean it should be left out of a language? I don't think so, but perhaps a language (like Ada) not aimed at prototyping situations could provide compiler support for restricting inheritance to subtype relationships. Also, I suspect that Smalltalk is not quite so badly off in this regard as those languages with multiple inheritance, where misuse is that much easier and can get that much worse.

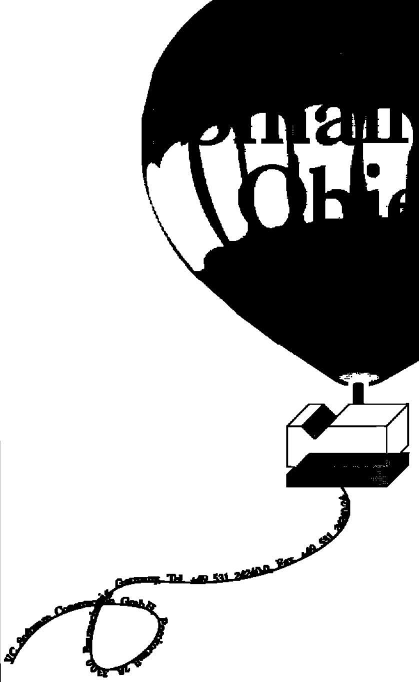
We haven't even begun to talk about dynamic binding, which could easily fill another whole column. In the context of these kind of discussions, full dynamic binding in the style of Smalltalk is generally considered too dangerous, because it is possible to have errors which will not be detected until runtime. I find it odd that people do not seem to have the same problem with current computer arithmetic systems, which have the potential to cause very serious runtime errors, but I guess they have different priorities. Even limited dynamic binding such that is found in C++ and Eiffel is considered dangerous. In a future column I may go into these issues, but I suspect most of the readers have already made up their minds on this and would be more interested in something else.

TO BE CONTINUED

This column is already running long and late, and I haven't even considered some of the more interesting criticisms and misperceptions. Among these are the idea that O-O implies that only local knowledge can be used, or that strict O-O does not permit classes representing relationships or processes. Those issues will have to wait for the next installment. ■

Alan Knight is a researcher in the Department of Mechanical and Aerospace Engineering at Carleton University, Ottawa, Canada, K1S 5B6. He currently works on problems related to finite element analysis in ParcPlace Smalltalk, and has worked in most Smalltalk dialects at one time or another. He can be reached at 613.788.2600 x5783, or by email as knight@mrco.carleton.ca.

ODBMS



ODBMS
The Objectoriented Database for Windows and OS/2.

ODBMS
To make your SMALLTALK even better.

ODBMS
Objectoriented Technology by VC Software Construction.

ORDER NOW !

- ☐ **ODBMS - Programmer's Version**
- ☐ **ODBMS - Graphical User Interface**
- ☐ **ODBMS - Interface to SQL**

- ☐ **DSSDe - Distributed Programming based on ODBMS**

ODBMS and more ...

USA: Power Marketing Inc., 101 Slough Road, Harvard MA 01451, Tel: 508-456-8302, Fax: 508-263-0696 <> UK: Cocking & Drury Software Ltd., 180 Tottenham Court Road, London W1P 9LE, Tel: 071-436-9481, Fax: 071-436-0524 <> VC Software Construction GmbH, Petritorwall 28, 3300 Braunschweig, Germany, Tel: +49-531-24 24 00, Fax: +49-531-24 24 0-24

Kent Beck

Abstract control idioms

I started writing about the new ValueModel style used in ParcPlace's Objectworks\Smalltalk release 4 as promised, but soon discovered that I need to cover some preliminary material about the "traditional" style first. I split the column into two parts. This one talks about how abstract control has been used to date. Next issue's will cover the new possibilities available with the advent of ValueModels.

MESSAGES LIMIT REUSE

Reuse is the name of the game. Headlines shout. Marketing literature trumpets. Salesmen ooze. Objects will solve your reuse problems. Not true, of course. Programmers solve reuse problems. It is possible to reuse procedural code, and it can be impossible to reuse objects. If the mere presence of objects doesn't enable reuse, what is it that makes reuse happen, technically?

Whenever I am able to reuse a piece of code, either by design or through serendipity, it is because the code makes few assumptions about what the rest of the world looks like. A graphics model that assumes all coordinates are integers is significantly harder to use than one that is prepared to take any kind of number. What does this have to do with messages limiting reuse?

Every time you send a message you build into your code the assumption that one and only one action will be invoked. What happens when you later decide you need two things to happen? Or sometimes none and sometimes many? You have to change the original code.

I can think of three levels of code reuse. By far the simplest is reuse by instantiation. You create an object, send it some messages, and good things happen. Far more complicated is reuse by refinement. To subclass you have to understand the inner workings of the superclass to know what messages to intercept and how to compatibly extend the representation. By far the most expensive reuse in terms of downstream costs is reuse by tweaking. Somehow the original author never factors the methods enough, but by a little judicious editing you can create an object you can subclass for your purposes.

Tweaking is becoming infeasible as a production programming strategy. As Smalltalk programs grow, it becomes increasingly desirable to treat code from outside sources as immutable. I have enough trouble keeping up with changes to my own objects, much less trying to track what several vendors have done with code I have modified. If we had a mechanism that was like message sending, but was extensible without modifying the original code, we could gain reusability for our libraries of objects.

THE SMALLTALK SOLUTION: UPDATE/CHANGED

Update/changed, also known as change propagation or dependency, is the Smalltalk solution to a "more abstract message send." It is more abstract in the sense that zero or more receivers can be activated by one action in the sender; the number and identity of the receivers is determined at runtime and can easily be changed by code which is otherwise unrelated to the sender; and the receiver has much more choice in responding to the a changed message than an ordinary message send. On the other hand, because it is not implemented by the Smalltalk virtual machine it is not as efficient as ordinary message sending.

I talked to Diana Merry-Shapiro (one of the long-time members of the original Smalltalk team) about the evolution of the dependency mechanism. The early Smalltalkers took as their benchmark problem a model consisting of a collection of numbers and two views, one a pie chart and the other a bar chart. The problem was to keep both charts consistent with the model while leaving the model as ignorant of the fact that it was being viewed as possible. According to Diana, it was Dan Ingalls who finally implemented the dependency mechanism as we know it.

Here is a quick review of the fundamentals of dependency. The system associates with each object a collection of dependents, other objects to be notified when it changes. Here is a simplified implementation:

```
Object>>addDependent: anObject
        Dependents "a class variable in Object" isNil
                ifTrue: [Dependents := IdentityDictionary new].
        (Dependents includesKey: self)
                ifFalse: [Dependents at: self put: Set new].
        (Dependents at: self) add: anObject
Object>>removeDependent: anObject
        Dependents isNil ifTrue: [^self].
        (Dependents at: self ifAbsent: [^self])
                remove: anObject
                ifAbsent: [^self].
        (Dependents at: self) isEmpty
                ifTrue: [Dependents removeKey: self]
```

Most objects don't have any dependents, and there is no space cost for nonparticipants, so the memory overhead of dependency is not high.

When an object changes its state in a way that it thinks dependents might be interested in it sends itself the message changed, which causes all of the dependents to be sent the message update. Each dependent then takes whatever action is necessary to reconcile it with the new state.

```
Object>>dependents
        Dependents isNil ifTrue: [^#()].
        ^(Dependents at: self) ifAbsent: [#()]
Object>>changed
        self dependents do: [:each | each update]
Object>>update
        ^self "Do nothing by default"
```

The solution to the benchmark problem mentioned above is to make the pie chart and the bar chart dependent on the list of numbers. Every time the list changes, adds, or deletes a value, it sends itself a changed message. Both of the views in their update methods simply redisplay and the consistency problem is solved. The solution has the additional attraction that new kinds of views can be added, and as long as they are registered to the model they will operate without any changes to the model. Finally, the model works in the absence of a user interface just as well as it does interactively. Because all communication with the user interface is through dependency, its presence or absence makes no difference to the model.

The first problem that becomes apparent with this simple dependency mechanism is that not every dependent is interested in every change. The most common form of changed message adds a parameter, a symbol by convention, which suggests the kind of change taking place. The parameter is passed along to the dependent. Notice that the generic update gets sent if update: is not overridden.

```
Object>>changed: aSymbol
        self dependents do: [:each | each update: aSymbol]
Object>>update: aSymbol
        self update
```

Most applications that need dependency can be coded with no more complexity than this.

DEPENDENCY IDIOMS

For consumers of update messages, the primary idiom is to override update: and switch on the parameter. Here is ListView>>update:

```
update: aSymbol
        aSymbol == #list
            ifTrue: [^self setNewList].
        aSymbol == #listIndex
            ifTrue: [^self setNewSelection]
```

I have found it good practice to have only a single send to self for each case. When I am tempted to put several statements in the block I invariably end up creating a method later which is exactly those lines. Also, the code is simpler to read if each implementation of update: has the same form.

What if you want several views on the same model, but you want each to respond to different updates? The old license version 2 image introduced pluggable views to solve this problem. Rather than create a subclass for each slight variant, each of which would override update:, a pluggable view stores the pattern against which update messages are matched in an instance variable. Here is SelectionInListView, the pluggable variant of ListView.

```
update: aSymbol
        aSymbol == partMsg
            ifTrue: [^self setNewList].
        aSymbol == initialSelectionMsg
            ifTrue: [^self setNewSelection]
```

The instance variables are set when a list is created with the SelectionInListView>>on:aspect:blah:blah: message. Each list also needs to send a different message to the model to get the contents and set the selection. The symbols used for checking updates double as messages that are sent to the model via perform:. I have always thought this was kind of sleazy, but in practice it works quite well.

The other commonly used pluggable view is TextView. SelectionInListView uses one symbol to check to see whether to update the list contents and another as the message to send to the model to get the list. TextView uses the same symbol for both (the aspect: parameter of the instance creation message).

A final note about implementing update:—remember to send "super update: aSymbol" if the current method hasn't consumed the update message. That way your classes will fit more neatly into hierarchies.

I looked through all senders of changed: to see if I could find any pattern to the symbols that are used as the parameter, and I wasn't able to discover anything profound. The parameter should, of course, have some relation to the change taking place in the model. Other than that there doesn't seem to be much of a pattern to how the symbols are selected.

DECIDING TO USE DEPENDENCY

If dependency is so cool why not use it all the time? Playground, a language I worked on at Alan Kay's Vivarium project, was an attempt to do just that. It used dependency as its only control abstraction. Because Playground was a pure abstract control language, it threw the two biggest drawbacks of dependency into high relief: debugging and performance.

There are two problems with debugging update messages. The first is in the debugger. It takes a long time to single-step through code which does an update. You have to go through all the intermediate steps of the implementation for each dependent. (The real implementation is considerably more complicated than the one outlined above. See the section called Gory Details for the, well, you know.) If you have lots of dependents and only one of them is interesting this can be tedious and frustrating.

The browser also does little to help debug dependency. If you have symbols built in to your implementations of **update:** you can at least use senders (from the **Launcher** window) to find out where they are used as parameters to **changed:**. If you are implementing a pluggable view, however, the symbol will only show up in the user interface code which creates the view. From this it is often hard to see how an update will be triggered. A trick I use is to add **"Transcript cr; show: aSymbol"** as the first line of the **update:** method I am interested in. I can then see all the update messages and the order in which they arrive.

A less compelling, but occasionally fatal, drawback of dependency is performance. Unlike a message send, which every part of the Smalltalk implementation is tuned to make efficient, changed messages have to go through several layers of invocation to get to their recipient. If you have lots of dependents, most of whom aren't interested in most updates, you can spend enormous amounts of effort creating a little activity. A related minor annoyance is that all those layers of invocation tend to clutter performance profiles, especially if you have several layers of updates happening.

Since dependency has significant costs associated with it, when is it worth using? The one clear case is when you are implementing new views or models. You need dependency so your code fits well with the rest of the system. Also, dependency makes your models more reusable by insulating them from the precise details of the interface or interfaces that are viewing them.

Other than models and views in the traditional sense, you should use dependency anywhere you want an object to be thoroughly insulated from the environment in which it operates. Any object that you know will be used in a variety of ways and that you want to keep clean is a candidate for dependency.

When is dependency being abused? Here are some signals that you have gone too far:

- An action spawns several updates and their order matters
- You forget which symbols mean what
- Your update messages create an infinite loop
- You find update messages that aren't handled by anyone

When your code begins exhibiting any of these symptoms, it is time to revisit the decision to use dependency. You may discover that one of the connections you are making always works out to use exactly one object, in which case you can replace the dependency with a direct reference and message sends. Or you may have a collection of objects that all respond to the same messages, so you can store a collection and use direct messages.

THE GORY DETAILS

The dependency implementation in Objectworks\Smalltalk release 4 is more complicated than the one outlined above. There is a variant of the update method that takes three parameters: an aspect, an optional parameter, and the changing object. **Changed:** sends **changed:with:**, which sends **update:with:from:** which by default sends **update:with:** which sends **update:**. All of these intermediate steps add greatly to the functionality and complexity of dependency. However, in my opinion, if you use all the available generality of the three-parameter version of **update:** you are stressing what was intended to be a very simple mechanism, and you are likely to run into trouble.

The implementations of **addDependent:** and **removeDependent:** in **Object** are much like the ones above. They have a serious flaw. If an object has been registered as a dependent and it fails to remove itself, or if an object gains dependents that are not removed, it cannot be garbage collected because it is referred to from a global variable. To deal with this problem, there is a subclass of **Object** called **Model** which adds an instance variable, dependents, and overrides **addDependent:** and **removeDependent:**. Since the model is not referred to globally, it is easier to get it garbage collected; once it has been collected, it no longer refers to its dependents, so they become candidates for collection.

A final nuance of the implementation of dependency is the use of **DependentsCollection**, a subclass of array. If a **Model** has only a single dependent the value of its instance variable **dependents** is that dependent. **Object>>changed:with:** sends **update:with:from:** and off to that dependent and everything works. If there is more than one dependent then dependents is a **DependentsCollection**, which overrides **update:with:from:** to forward the message to each of its elements. This little trick saves an additional object when there is only one dependent.

CONCLUSION

We have seen how abstract control structures, implemented by Smalltalk dependency mechanism, can reduce the strength of the connection between two objects. This can lead to enhanced reusability. Because it is outside the language and is not directly supported by the programming environment, excessive use of dependents can make programs hard to read and debug, and can lead to performance problems. ■

Kent Beck has been discovering Smalltalk idioms for eight years at Tektronix, Apple Computer, and MasPars Computer. He is also the founder of First Class Software, which develops and distributes re-engineering products for Smalltalk. He can be reached at P.O. Box 226, Boulder Creek, CA 95006 or kentb@maspar.com.

Juanita Ewing

Creating subclasses

Class hierarchies are a way to capture variations and specializations. A subclass is generally a more specialized kind of entity than its superclass. For example, the class Sphere is a subclass of the class Solid. If we needed a representation for pyramids, we would create a new subclass, Pyramid, whose superclass is Solid.

In this example it is easy to decide how Pyramid fits into the hierarchy because there is an abstract superclass. This abstract superclass is a generalization representing different kinds of solids. Often the decision about where to insert a new class in the hierarchy is not straightforward. This column explores strategies for placing subclasses in a hierarchy and consequences of the placement.

BENEFITS

Well-formed class hierarchies are those in which functionality is factored into a number of classes. Subclasses are specializations, and superclasses are generalizations. When functionality is factored into hierarchies, classes are more reusable and maintainable. Highly factored hierarchies are also easier to extend.

HEURISTICS

A significant part of creating subclasses is choosing the most appropriate superclass. It is almost always better to inherit behavior rather than reimplement behavior, though not at the cost of inheriting inappropriate behavior. In order to inherit the greatest amount of appropriate behavior, we use two heuristics to select candidate superclasses.

HEURISTIC ONE

Look for a class that fits the "is a kind of" or "is a type of" relationship with your new subclass. Often it helps to make this heuristic into an English question. For example, we can ask the question, "Is a pyramid a kind of solid?"

Documentation describing a class often helps you understand exactly what the class represents. Because of your understanding of classes that you implemented, it is much easier to insert new classes into hierarchies that you have developed. Personal knowledge of the class hierarchy can substitute for class documentation.

HEURISTIC TWO

Look for a class with behavior that is similar to the desired behavior of the new subclass. In this heuristic you must look at the methods or good documentation for the methods. Often, just the message selectors will give you enough information to reject many inappropriate classes.

BEHAVIORAL INHERITANCE VS. IMPLEMENTATION INHERITANCE

The two heuristics we have presented are oriented toward class hierarchies based on behavior. This kind of inheritance is known as *behavioral inheritance*. In these hierarchies a subclass and its superclass have a subtype relationship. That is, the subclass supports all the behavior that the superclass supports, and the subclass can add new behavior. Any use of an instance of the superclass can be replaced by the use of an instance of the subclass. Some examples from Smalltalk class libraries are: RecordingPen is a subclass of Pen, Time is a subclass of Magnitude, Integer is a subclass of Number, and WildPattern is a subclass of Pattern.

Inheritance can also be used in a more pragmatic fashion, in which a class is placed in a hierarchy because of the desire to inherit code and implementation rather than behavior. Inheritance used in this fashion is called *implementation inheritance*. Most class libraries also have examples of this kind of inheritance: Process is a subclass of OrderedCollection, and Debugger is a subclass of Inspector.

BUSROUTE EXAMPLE

An example involving bus routes will illustrate the different kinds of inheritance. In this example, we need to create a class to represent a bus route, which is used to inform the bus driver and passengers of the bus' path through the city. A bus route is a collection of bus stops, in a particular order. A bus route needs the ability to compose the route out of bus stops, to supply a summary report on the route's stops, to determine how many intermediate stops there are between two stops, and the fare from one stop to another. The fare computation may vary depending on which zones the stops are located in.

People who are familiar with Smalltalk class libraries will immediately start to think of the class OrderedCollection when they read the description of a bus route. OrderedCollection is a concrete collection class that holds elements in order, similar to a stack or queue. The elements can be of any type.

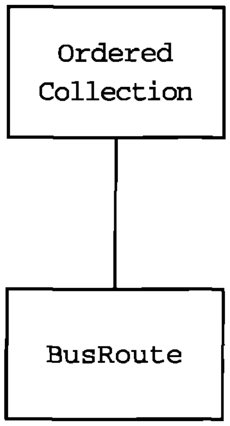


Figure 1. BusRoute as a subclass of OrderedCollection.

IMPLEMENTATION INHERITANCE ALTERNATIVE

We need to make a new class, which we will call BusRoute. Should BusRoute be a subclass of OrderedCollection? As a subclass of OrderedCollection, it would inherit the implementation that maintains elements in order. It would also inherit the code for adding and removing elements which can be used to compose the bus route. This relationship is shown in Figure 1.

It is useful to determine whether this placement of BusRoute uses behavioral inheritance or implementation inheritance. Is a bus route a kind of ordered collection? No. Instances of OrderedCollection have an implicit responsibility to hold objects of arbitrary type, and a bus route holds only bus stops. A BusRoute is not a generic data structure class.

Is all the behavior of OrderedCollection appropriate for BusRoute? No. According to the description, bus routes shouldn't respond to the do:, select: or reject: messages, or many of the other generic collection messages. Therefore, BusRoute is not a subtype of OrderedCollection. Placing BusRoute as a subclass of OrderedCollection is an example of implementation inheritance, in which code and implementation are usefully inherited.

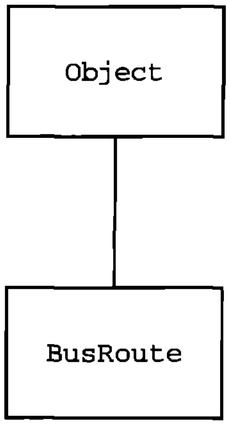


Figure 2. BusRoute as a subclass of Object.

BEHAVIORAL INHERITANCE ALTERNATIVE

Another alternative is to make BusRoute a subclass of some other class. A bus route is a kind of route. Are there any route classes in the Smalltalk library? If the answer is no, then make BusRoute a subclass of Object. The behavior of Object is appropriate for all objects, so Object is selected when there isn't any other appropriate superclass. This alternative is an example of behavioral inheritance because all the behavior in Object is appropriate for BusRoute. The inheritance relationship is shown in Figure 2.

In this alternative, BusRoute would collaborate with OrderedCollection to store bus stops in order. Figure 3 illustrates the collaboration between the two objects. An instance variable, busStops, references an instance of OrderedCollection that stores bus stops. Instances of BusRoute can relay messages to the instance of OrderedCollection referenced by the busStops instance variable.

OVERRIDE INAPPROPRIATE METHODS

In the first alternative, in which BusRoute was a subclass of OrderedCollection, we proposed using inherited public methods such as add: and remove: to compose the bus route. But this is not a very good way to compose bus routes because bus routes would be subject to accidental and inappropriate modifications. Further, if a bus stop is added to a route, then what results is a new and different route. It should not be the same object.

Many methods must be overridden to disallow in-place modifications. For example, the add: method is public and should be overridden to prevent changes.

```
BusRoute subclass of OrderedCollection

instance methods

add: aBusStop
        "Override inherited public method to produce an error.
        Bus stops cannot be added to a route."

        ^self error: 'Bus routes cannot be modified.'
```

In the second alternative, in which BusRoute is a subclass of Object, we don't need extra methods to override inappropriate behavior.

CREATE NEW BUSROUTES

A more appropriate way to compose bus routes disallows in-place modifications. We need to make an instance creation

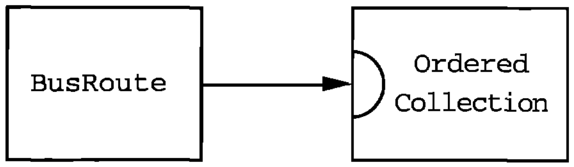


Figure 3. BusRoute collaborates with OrderedCollection.

method that creates an initialized bus route. To support the instance creation method, a private instance method is needed to set the collection of bus stops.

```
BusRoute subclass of Object

class methods

withAll: collectionOfBusStops
    "Create a new instance of the receiver initialized
    from <collectionOfBusStops>."

        ^self new busStops: collectionOfBusStops

instance methods

busStops: collectionOfBusStops
    "Private - Set the collection of bus stops."

        busStops := collectionOfBusStops
```

Classes that collaborate with BusRoute need to access the bus stops to select stops based on some criteria. The class BusRoute needs to provide access to the bus stops *and* protect the private collection of bus stops from modification. The instance method busStops returns a copy of the collection referenced by the instance variable. This way collaborators can modify the returned collection of bus stops without any side effects on the bus route.

```
BusRoute subclass of Object

instance methods

busStops
  "Return a copy of the collection of bus stops."

  ^busStops copy
```

New bus routes can be created using these methods. The following code illustrates the creation of a new route based on the bus stops from another route:

```
shoppingStops := downtownRoute busStops.
shoppingStops removeFirst.
derivedRoute := BusRoute withAll: shoppingStops
```

REUSE IMPACTS

One of the benefits of the behavioral inheritance alternative is that it is easy to change the collection characteristics. It is easier to modify the initialization code that allocates an object for an instance variable than to rearrange the hierarchy in order to get different collection characteristics. If you are forced to rearrange the hierarchy, then collaborators of BusRoute must change also. This is because different collection classes respond to different messages, and the inherited messages are directly accessed by the collaborators of BusRoute.

Also, new subclasses of BusRoute can be created based on collection characteristics. One subclass could support set semantics, in which no duplicates are allowed. Another could support sorted collection semantics. Each subclass can be implemented by simply overriding the initialization method. This is much more awkward with implementation inheritance.

In the behavioral inheritance alternative, the exact collaboration between BusRoute and OrderedCollection is clear because messages are relayed to the instance of OrderedCollection from BusRoute. In the implementation inheritance alternative, there is no collaboration. An examination of BusRoute does not determine which methods from OrderedCollection are used by BusRoute. Instead, collaborators of BusRoute must be examined. Furthermore, because not all inherited messages are appropriate, other developers will not know which messages they can send to BusRoute. This makes it much more difficult to extend and maintain the application containing BusRoute, and to reuse BusRoute in related applications.

Zippy Object Oriented Memory™

Object Oriented Database Management System

The ONLY ODBMS for Smalltalk for under $1000 that delivers Persistent Object Storage on Disk via a Zippy B+Tree Database Retrieval Engine!

ZOOM

for Smalltalk/V and Smalltalk-80
All Platforms $199.95
Source Code Included

Hierarchical Applications Limited (612) 837-2117
12407 Mopac Expwy N., Suite #100-266
Austin, TX 78758

SUMMARY

Use behavioral inheritance whenever possible, because the resulting subclasses will be more reusable and easier to maintain. In locating subclasses in a hierarchy, use the is-kind-of criteria and similar behaviors to guide your selection. Only after locating a subclass based on behavior should you examine implementation details.

It is okay to change the superclass. After some implementation and testing, it is quite common to revisit class placement in the hierarchy. Reexamining placement can occur in conjunction with reorganizing the entire hierarchy and with the addition of new classes. ■

Juanita Ewing is a senior staff member of Digitalk Professional Services (formerly Instantiations Inc.). She has been a project leader for several commercial object-oriented software projects, and is an expert in the design and implementation of object-oriented applications, frameworks, and systems. In a previous position at Tektronix Inc., she was responsible for the development of the class libraries for the first commercial-quality Smalltalk-80 system. Her professional activities include Workshop and Panel Chairs for the annual ACM OOPSLA *conference.*

LAB REPORT

Justin O. Graver

Smalltalk research at the University of Florida

Reports of current work in Smalltalk taking place in leading university and research laboratories.

The main focus of Smalltalk research at the University of Florida is to demonstrate the viability of evolutionary prototyping as an alternative to the traditional waterfall model of software development. Specifically, we are working on a variety of enhancements to the Smalltalk program development environment (PDE). These include new change management tools and techniques, application modules, and a string-to-object translator generator. We are also working on the more abstract tasks of understanding, formalizing, and validating evolutionary prototyping as a new software development methodology.

Smalltalk was originally designed as a programming environment for a single user. As Smalltalk moves slowly into the arena of large-scale industrial software applications produced by tens (or hundreds) of programmers, the need for more powerful change management and versioning capabilities is apparent. The change management tools project is addressing this need in a variety of ways. As a prelude to using a commercial database for source code management, we have broken up Smalltalk's monolithic source code file into several smaller module files. We've built tools for creating, loading, versioning, and maintaining large code libraries of module files. These tools automatically determine the complex dependencies that exist between library modules so that modules get loaded in the correct sequence. We have also chosen to experiment with the class as the smallest unit of versioned granularity (all change management tools available for Smalltalk PDEs that we know of use individual methods as the lowest level of granularity). Since large Smalltalk applications easily grow to include hundreds of classes with tens of thousands of methods, we felt that classes might provide a more convenient and manageable level of granularity.

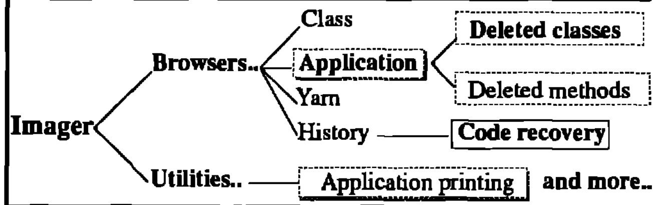
Closely coupled with the change management improvements is the notion of application modules. There is a widely recognized need for separate name spaces within a single Smalltalk image. Our application modules provide user-defined statically scoped name spaces. This permits enforceably private classes and the specification of well-defined module interfaces. We also plan to implement statically enforceable private methods. The ultimate goal of the module system is to support modules with statically typed public interfaces. These typed interfaces would allow each module to be independently compiled and optimized, using the TS (Typed Smalltalk) compiler. We have designed and prototyped a typed module compiler. Given type declarations for only public interface messages and primitives, the module compiler statically determines all intermediate type information necessary to compile and partially optimize the module (further optimization can be performed when different modules are linked together). The module compiler uses a technique called abstract interpretation to trace the effects and requirements of each public interface message. This technique also handles arbitrarily recursive methods. To provide better support for both TS and the module compiler, and to fuel additional research into object-oriented compiler research, we have designed and implemented a string-to-object translator generator tool called T-gen. T-gen provides a comprehensive set of translator generator options including EBNF grammar specification, fully automatic keyword detection, tracing and consistency checking, LL and LR parser generation, support for automatic generation of parse trees, and automatic generation of T–gen-independent scanner and parser classes.

A Smalltalk parser (to be used as the new TS front end) is just one of the projects that has been successfully built using T-gen. We have also used T-gen to implement a compiler for an instructional language called Tiny. Tiny has been used for over five years in the graduate and undergraduate compiler courses at the University of Florida. We hope to use the new Smalltalk implementation of Tiny to further instill object-oriented principles into the CIS curriculum. These research projects have been primarily funded by the Software Engineering Research Center (SERC). SERC is an NSF sponsored Industry/University Cooperative Research Center. Researchers from Arizona State University, Purdue University, and the University of Florida are working jointly with over a dozen major companies to solve current problems in software engineering. For more information about SERC contact Tammera Reedy by phone at 904.392.1520, or by e-mail at tcr@ufl.edu. Much of this research is an outgrowth of the ongoing Typed Smalltalk project at the University of Illinois at Urbana-Champaign. We maintain active collaboration with that team, which is lead by Ralph Johnson. This successful interaction has been greatly facilitated by the sharing of tools developed at both institutions. T-gen runs under ParcPlace's Objectworks/Smalltalk Release 4 and is available via anonymous ftp from the University of Illinois Smalltalk Archives (st.cs.uiuc.edu) in the directory /pub/st80_r4/T-gen2.0. ■

Justin Graver, Ph.D., is currently a staff engineer at Motorola's Software Technology Center where he is working on next-generation CASE technology using Smalltalk. Prior to that, he was an assistant professor at the University of Florida. He can be reached at Motorola Software Technology Center, 1301 E. Algonquin Rd, Schaumburg, IL 60196, by phone at 708.576.1916, or by email at graver@comm.mot.com.

SixGraph™ Smalltalk/V users: the tool for maximum productivity

- Put related classes and methods into a single **task-oriented object** called application.
- **Browse** what the application sees, yet easily move code between it and external environment.
- Automatically **document** code via modifiable templates.
- Keep a **history** of previous versions; restore them with a few keystrokes.
- **View** class hierarchy as graph or list.
- **Print** applications, classes, and methods in a formatted report, paginated and commented.
- **File** code into applications and **merge** them together.
- Applications are unaffected by **compress log change** and many other features..



CodeIMAGER™ V286, VMac $129.95
VWindow & VPM $249.95
Shipping & handling: $13 mail, $20 UPS, per copy
Diskette: ☐ $3^{1/2}$ ☐ $5^{3/4}$

SixGraph™ Computing Ltd.
formerly ZUNIQ DATA Corp.
2035 Côte de Liesse, suite 201
Montreal, Que. Canada H4N 2M5
Tel: (514) 332-1331, Fax: (514) 956-1032
CodeIMAGER is a reg. trademark of SixGraph Computing Ltd.
Smalltalk/V is a reg. trademark of Digitalk, Inc.

PRODUCT ANNOUNCEMENTS

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

Synergistic Solutions Inc. has announced additional platform support for **Smalltalk\SQL**, the portable database interface for Smalltalk. The product works in conjunction with the latest releases of ParcPlace Systems' Objectworks\Smalltalk and Digitalk Smalltalk/V. The product enables development of graphical user interface (GUI) applications which access information stored in relational databases.

Smalltalk\SQL provides direct Sybase connectivity for Windows 3.0, Macintosh, Sun, RS/6000 and other UNIX platforms. Direct Oracle support is currently available for Windows 3.0 and Sun SPARCstations. Gupta and NetwareSQL support is available for the Windows 3.0 and OS/2.

For more information, contact Synergistic Solutions Inc., 63 Joyner Dr, Lawrenceville, NJ 08648, 609.586.0025.

Logic Arts announces **VOSS/Personal**, two low-cost versions of the Virtual Object Storage System for Smalltalk/V 286 and Smalltalk/V Windows.

VOSS/Personal is fully compatible with the equivalent main product line, and can read and write the same virutal object spaces, providing transparent access to persistent Smalltalk objects of any class on disk, without the need for a separate DBMS programming language. It has the same transaction management of updates, the variable-size cache of virtual objects in the image, and most of the same VirtualDictionary and Virtual Collection classes for managing collections larger than the image.

For more information, contact Logic Arts Ltd, 75 Hemingford Rd, Cambridge, England, CB1 3BY, +44 223 212392, fax +44 223 245171.

Servio Corp. and **Hewlett-Packard Co.** announced that Servio has been named an HP Value-Added Business Partner and that Servio's **GemStone object database** and **GeODE object development environment** will be made available for the HP Apollo 9000 Series 700 PA-RISC-based workstation family in the third quqrter of 1992.

GemStone is the only ODBMS to support applications written in C, C++, and Smalltalk. GemStone's Object Development Environment, GeODE, is the first code-free development environment for visually and graphically designing and building ODBMS applications.

For more information, contact Servio Corp., 950 Marina Village Parkway, Suite 110, Alameda, CA 94501, 510.814.6200.

VC Software Construction has announced enhancements to their object-oriented database management system, **ODBMS**. The package supports most Smalltalk languages. Its storage facilities of objects can be used during the development of Smalltalk applications as well as by a standalone database application. ODBMS stores items in opposition to relational databases' arbitrary complex data types. There is almost no limitation to the structure and length of these items.

ODBMS/SQL uses the optimized query algorithms of SQL to retrieve objects faster. The integration of the access to relational databases into ODBMS avoids redundancy in stored items and enhances the use of Smalltalk in the commercial environment.

For more information, contact: VC Software Construction, Petritorwall 28, 3300 Braunschweig, Germany, +49 531 24 24 00, fax +49 531 24 24 0 24.

Highlights

Excerpts from industry publications

WORDS OF WISDOM

...Can you say object-oriented?...How about real-time systems, graphical environments, multimedia, or CASE technology? If you want to get laid off, don't mention or learn these topics. Even better, tell anyone who will listen that object-oriented development is only a fad...

Eleven ways to get laid off, Karen Hooten, ***COMPUTER LANGUAGE,*** *3/92*

SMALLTALK

...You must realize that I use a variety of languages in my work: C++, Smalltalk, and Ada, in particular. I use Smalltalk for prototyping, and here, its dynamic binding allows me to throw together prototypes quickly, with blatant disregard for any kind of type of safety or robustness. For the kind of experimental development I do, this is precisely the kind of flexibility I need...

Interview with Grady Booch, ***THE C++ JOURNAL,*** *vol.2/no.1, 1992*

STRATEGIES

...OOP-based software will present courts with challenging questions concerning, among other things, infringement and risk allocation. As the courts wrestle with these issues, suppliers, developers, and users must be careful that agreements with one another address, to the extent possible, their specific rights and duties in this changing area.

Making sure that OOP doesn't become oops, Robert V. Hawn, ***BUSINESS JOURNAL SERVING SAN JOSE AND THE SILICON VALLEY,*** *3/16/92*

MULTIMEDIA

SimGraphics Engineering Corp. is changing the face of animation. Using a powerful graphics workstation, a face armature, an object-oriented toolkit, and one of the world's most famous software game characters, the company is ushering in the day when real-time animation will largely replace frame-by-frame animation. "Obviously, you always will be able to get higher resolution from frame-by-frame animation, but there will be a point when both software and hardware will permit most of the animation that now is being done frame by frame to be done in real time," says Steve Glenn, vice president of New Business Development for SimGraphics of South Pasadena, CA...

The many faces of Mario, Margaret Seaborn, ***WORKSTATION NEWS,*** *5/92*

...Object-oriented programs already exist for imaging, though they aren't well publicized. This is unfortunate, as object-oriented programming will affect the growth of digital photography more than anything else...

Digital photography: changing for the better, John Larish, ***PHOTO ELECTRONIC IMAGING,*** *4/92*

...For multimedia computing, all the Dataquest survey respondents felt there was still a lot to be done in providing application software, increasing network data rates, providing wideband telecommunication networks, and finalizing standards. But despite these unresolved problems, the multimedia juggernaut rolls on. "To many, this looks like a tidal wave starting to swell," says AT&T's [Arnold] Englander. Certain elements are in place, he says: the readiness of the telecommunications infrastructure; the emergence of critical video-compression and telecom standards; advances in image compression, VLSI technology, and object-oriented programming; and the high costs of travel in a business environment that's ever more global in scope. More rapidly than expected or imagined, the elements that must combine to make multimedia a reality are coming together. And "the ambitions of diverse businesses competing and cooperating in the convergence of telecommunications, computing and TV," as Englander puts it, guarantee that the ride will be an interesting one.

En route to collaborative computing, Samuel Weber, ***ELECTRONICS,*** *4/92*

CREATIVE IMPLEMENTATION

...If no one knows what is going on inside an object's functions, and no one can tamper with its data without authorization, then an object is highly secure. It polices its own borders, responding only to authorized messages...Since an object has boundaries, you can own it. You can reward or punish the persons who designed it. You can rent out the use of the object without telling how it works. You can see a certain appeal here to the corporate mind...

Object-oriented programming: what's the big deal?, Birrell Walsh, ***MICROTIMES,*** *3/92*

...I've discovered that the single greatest challenge of tackling a new object-oriented program is keeping a vision of the program that's accomplishable. As I was writing this program, I have to admit that at times I was thinking of an interactive CD-ROM-based multi-media extravaganza. Luckily, common sense and deadlines prevailed...

Expert's toolbox: templates of doom, Larry O'Brien, ***THE CHICAGO PURCHASER,*** *5/92*

TOOLS AND LIBRARIES

...There's much more to realizing the benefit of a class library than simply buying one at random and throwing it at a development problem (or team). There are three general problems that can make it difficult to make good use of class libraries. First, since you're expected to derive new classes, what's to prevent you from creating a mess?...A second problem occurs when you try to incorporate a class library into an existing application. You may have an optimal application, but the library designer had a unique purpose in mind: to create the optimal library design. Are the two designs compatible?...Finally, different vendors may have differing ideas about optimal library design.There isn't really any such thing as a standard for class libraries...All these problems have one trait in common: inconsistency. None of the problems are really the fault of class libraries per se; it's really the way we create and use them along with our own understanding of the proper approach to object-oriented program development that determines whether class libraries are a major benefit...

Development tools, Mike Stewart, ***COMPUTER SHOPPER,*** *3/92*

...In the future, access to object libraries may determine which developer (or company) is able to serve clients most adequately. It is hoped that the elegance of one's code, long a measure of the quality of one's product, will remain the determinant of success in our industry. However, this may be the case only if no one company or class of companies is able to dominate the source of software objects. In a perfect world, there will be a plentiful supply of public domain objects accessible to everyone, via the same channels from which we are all now accustomed to getting our sources.

Concerning your career, Jim Johnson, ***UNIFORUM MONTHLY,*** *3/92*

ANALYSIS AND DESIGN

...[Rob Dickerson, VP and General Manager of The Database Business Unit at Borland International]: I think you've got to learn to do a class hierarchy. The first time you do your class hierarchy, you write out what looks obvious, and you fiddle with it, and you realize it's not the best one. So you redesign it, and by the time you're done, the class hierarchy you end up with was not what you initially thought. And there's a bunch of tricks to it—how to identify a meta-class, factoring, the notion of collection classes, how to design a class hierarchy, but that's the main design effort. At least, that's what I've seen our R&D guys have to get their hands around. [Jacob Stein, Chief Technologist for Servio Corp]: And there's lots of trade-offs, trade-offs between reusability, and a natural fit to the system you're modeling. They might not always be exactly the same. There may be a trade-off between designing for reuse and designing for this particular application, and you have to take that broader scope. It's said that people don't get classes right until they've been implemented about three times, which might mean that some of the interfaces will change during that course of time...

Roundtable: experts speak on object-oriented development!, John L Hawkins and Dian Schaffhauser, ***DATA BASED ADVISOR****, 4/92*

DISTRIBUTED ENVIRONMENTS

..."Object technology offers a second-generation model for client/server, with a clear role for a powerful client as well as a powerful server," said David Gilmour, executive vice-president of sales and marketing for Versant Object Technology, Menlo Park, Calif. By raising the power of an individual object to support transparent peer-to-peer communication via messages, the idea of client/server extends to a more robust notion of objects. Under this notion, objects could at one point make requests as clients to servers, then at other points act as server to other clients. This allows a modular distributed system that may be more responsive to change. Using objects as the unit to be distributed may allow developers to save implementation issues—such as distribution—until after the design is complete. "This is because object technology is an inherently parallel technology that naturally thrives in a distributed multiprocessing environment," said Dr. David Taylor, principal of Taylor Consulting, San Mateo, Calif...

Objects can set the stage, Eric Aranow and Tom Kehler, ***SOFTWARE MAGAZINE****, 5/92*

...Object-oriented DBMSs combine database technologies and object-oriented programming to provide greater modeling power and flexibility to programmers of data intensive applications. Over the last five years, OODBMSs have been the subject of intensive research and experimentation, which led to an impressive number of prototypes and commercial products. But the theory and practice of developing distributed OODBMSs have yet to be fully developed. Distributed environments will make the problems even more difficult. In addition, the issues of data dictionary management and distributed object management have yet to be dealt with. However, distribution is an essential requirement, since applications that require OODBMS technology typically arise in networked workstation environments...However, distributing an object-oriented database within a network of workstations (and servers) is becoming very attractive. In fact, some OODBMSs already support some form of data distribution transparency...

Distributed database systems: where were we?, M. Tamer Ozsu and Patrick Valduriez, ***DATA BASE PROGRAMMING & DESIGN****, 4/92*

As if the jump to a client/server information system paradigm were not tough enough, many companies are looking at moving to object-oriented programming (OOP) as well. By my measure, the OO market today is about where the client/server market was three to four years ago, and the two are even starting to merge in some areas.

VOSS

Virtual Object Storage System for

Smalltalk/V

Seamless persistent object management for all Smalltalk/V applications

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

Some comments we have received about VOSS:

"...clean ...elegant. Works like a charm."
–Hal Hildebrand, Anamet Laboratories

"Works absolutely beautifully; excellent performance and applicability." *–Raul Duran, Microgenics Instruments*

Logic ARTS

VOSS/286 $595 (Personal $199), VOSS/Windows S750 (Personal $299) (Personal versions exclude items marked (np)).
Quantity discounts from 30% for two or more copies. (Ask for details)
Visa, MasterCard and EuroCard accepted. Please add $15 for shipping.
Logic Arts Ltd 75 Hemingford Road, Cambridge, England, CB1 3BY
TEL: +44 223 212392 FAX: +44 223 245171

They are complementary technologies that, when combined, can give a company a formidable competitive advantage...

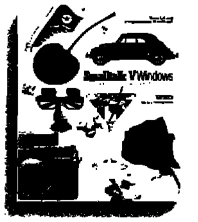
On the front end: Report card on Enfin/2, Robert C. Bolt, ***DBMS****, 4/92*

DATABASES

...In the world of textual data, relational databases worked fine. Text gives you structure and form in the way of character strings and numbers. This is something an RDBMS can handle quite well. Unfortunately, when you start dealing with multimedia data types—where you have to deal with massive amounts of this data, many of them being object-based—an RDBMS falls flat. By contrast, object-oriented databases come out way ahead of RDBMSs when dealing with heterogeneous, complex data involved in complex relationships. More importantly, when you start getting applications designed to integrate these multimedia data types into their programs, it will be important for them to include, as a part of the applications, an object-oriented database to help them handle these new types of object based data. At first, you will see these object-based databases added to authoring products, then to presentation, drawing and desktop-publishing products. They will also become important to any word-processing and next generation on-screen document communications. Ironically, it will not be the traditional database suppliers that will help these independent software vendors use a database effectively in this multimedia-driven world. Even though they all have object-based databases in the works, unless they are able to perfect them soon and make them work harmoniously with their RDBMS programs of today, they could be left out in the cold. In the future, the database will be embedded in major applications so they can manipulate these stored images, video and sound and integrate them into the heart of the app. Whether anyone likes it or not, multimedia computing is going to revolutionize the way we use computers.

The soft view: multimedia simply spells a new digital data type, Tim Bajarin, ***COMPUTER RESELLER NEWS****, 4/20/92*

WINNER PC WEEK LABS SHOOT-OUT



INTERNATIONAL ALLIANCE IBM AD/Cycle

WINDOWS AND OS/2: PROTOTYPE TO DELIVERY. NO WAITING.

In Windows and OS/2, you need prototypes. You have to get a sense for what an application is going to look like, and feel like, before you can write it. And you can't afford to throw the prototype away when you're done.

With Smalltalk/V, you don't.

Start with the prototype. There's no development system you can buy that lets you get a working model working faster than Smalltalk/V.

Then, incrementally, grow the prototype into a finished application. Try out new ideas. Get input from your users. Make more changes. Be creative.

Smalltalk/V gives you the freedom to experiment without risk. It's made for trial. And error. You make changes, and test them, one at a time. Safely. You get immediate feedback when you make a change. And you can't make changes that break the system. It's that safe.

And when you're done, whether you're writing applications for Windows or OS/2, you'll have a standalone application that runs on both. Smalltalk/V code is portable between the Windows and the OS/2 versions. And the resulting application carries no runtime charges. All for just $499.95.

So take a look at Smalltalk/V today. It's time to make that prototyping time productive.

Smalltalk/V

Digitalk, Inc., 9841 Airport Blvd., Los Angeles, CA 90045
(800) 922-8255; (213) 645-1082; Fax (213) 645-1306

LOOK WHO'S TALKING

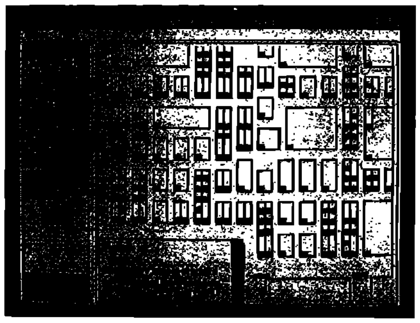
HEWLETT-PACKARD
HP has developed a network trouble-shooting tool called the Network Advisor. The Network Advisor offers a comprehensive set of tools including an expert system, statistics, and protocol decodes to speed problem isolation. The NA user interface is built on a windowing system which allows multiple applications to be executed simultaneously.

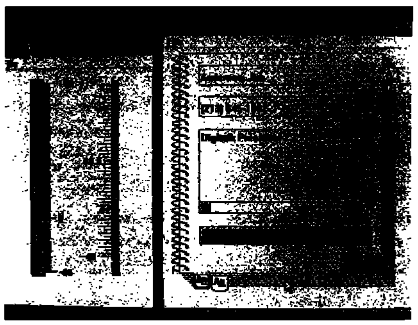
NCR
NCR has an integrated test program development environment for digital, analog and mixed mode printed circuit board testing.

MIDLAND BANK
Midland Bank built a Windowed Technical Trading Environment for currency, futures and stock traders using Smalltalk V.

KEY FEATURES

- World's leading, award-winning object-oriented programming system
- Complete prototype-to-delivery system
- Zero-cost runtime
- Simplified application delivery for creating standalone executable (.EXE) applications
- Code portability between Smalltalk/V Windows and Smalltalk/V PM
- Wrappers for all Windows and OS/2 controls
- Support for new CUA '91 controls for OS/2, including drag and drop, booktab, container, value set, slider and more
- Transparent support for Dynamic Data Exchange (DDE) and Dynamic Link Library (DLL) calls
- Fully integrated programming environment, including interactive debugger, source code browsers (all source code included), world's most extensive Windows and OS/2 class libraries, tutorial (printed and on disk), extensive samples
- Extensive developer support, including technical support, training, electronic developer forums, free user newsletter
- Broad base of third-party support, including add-on Smalltalk/V products, consulting services, books, user groups


This Smalltalk/V Windows application captured the PC Week Shootout award—and it was completed in 6 hours.


Smalltalk/V PM applications are used to develop state-of-the-art CUA-compliant applications—and they're portable to Smalltalk/V Windows.