The Smalltalk Report

The International Newsletter for Smalltalk Programmers

February 1992 Volume 1 Number 5

A PROPOSAL FOR CHANGE MANAGEMENT IN SMALLTALK

By Jeff McKenna

From the point of view of designing software, object-oriented techniques gain considerable advantage by reducing the semantic distance between the user and the programmer. Objects give us a way to talk about software using the same vocabulary as the end user.

Perhaps the same technique of involving the user can be employed when considering the problems of change management during object-oriented software development. In this case, the user is the software developer.

Good object-oriented programmers commonly use two distinct phases of development. The first is a functional expansion phase and the second is a "cleanup" or consolidation phase. What I propose is that a desirable change management system would acknowledge and support both of these phases. I think of these phases as natural events. They can be compared to breathing in and out, injecting life into the development process.

OBSERVING SMALLTALK PROGRAMMERS

When developing new software, good Smalltalk programmers explore the new functionality quickly by designing and implementing required new objects and by extending existing objects to support the new requirements. Typically, and correctly so, not much thought is given to issues of reuse or to the details of correct style.

After spending time on functional expansion, successful programmers will stop the expansion of functionality while they clean up their work. When pressed, they will say that they are improving understandability, readability, and reducing code bulk—"making it nice." At this stage, the hierarchy may be reworked as understanding of the application grows: Methods no longer required will be removed, method selectors will be renamed to more directly reflect their actual intent, and methods will be reworked to become more reusable.

One of the more interesting aspects of these observations is the timing of these two activities. Better programmers spend more time cleaning up than other programmers and they also tend to do it sooner. Typically, good programmers clean up after just a few days rather than weeks or months after development.

CHANGE MANAGEMENT

A primary role of change management in software development is to coordinate the work of multiple developers. In the wide open style of Smalltalk development, this usually means keeping the various individuals from stepping on each other's work. Management will also use the change management system to control releases, both internal releases for testing and external ones.

A typical approach for change management in Smalltalk is to assign owners to classes. Recently, we have seen some discussion of function-based ownership. Both of these approaches have advantages and disadvantages.

continued on page 4...

Contents:

John Pugh

Paul White

EDITORS' CORNER

Writing these notes just before heading off for New Year's celebrations spurred us to think of the major developments that have taken place in the Smalltalk world in the last twelve months. The language wars have continued, but C++ and Smalltalk have emerged as the dominant players as we enter 1992. Why has Smalltalk made the cut? We believe there are a number of reasons. The Smalltalk vendors have taken a number of steps to attract real developers by focusing on issues such as portability and providing better tools for application delivery. Third party vendors have addressed a number of areas critical to Smalltalk's acceptance such as user interface builders, access to databases, and performance profilers. The project management tools sorely lacking in the base Smalltalk systems have now appeared on the market in flavors to suit both small development teams and large groups of Smalltalk programmers accessing a shared code repository over a network. As the head of one major development group at a big multinational company told us recently at a conference, "Smalltalk went from having next to no support for project management to having perhaps the best object-oriented project management tools in the industry." Finally, we can point to documented Smalltalk success stories such as those reported on in the Experience Reports section of the October OOPSLA conference in Phoenix.

Where is Smalltalk heading in 1992? The MIS marketplace for one! That's right, the world of mainframes, Fortune 500 companies, and hundreds of thousands of lines of COBOL code has been taking a serious look at O-O technology and Smalltalk. They are still separating out the hype from the substance, but many forward-looking companies are already evaluating the technology through pilot projects with a considerable number choosing Smalltalk as the initial vehicle of choice.

In this month's lead article, Jeff McKenna continues our theme on change management. Following recent articles by Juanita Ewing and S. Sridhar that focused on class ownership, Jeff puts forward his view that change management is best organized around two distinct phases of software development using Smalltalk—functional expansion and consolidation phases. Also, for those Smalltalk programmers who still find occasion to lament the absence of a case statement in Smalltalk, Charles-A. Rovira provides the answer in his article—"A case in point."

Three of our regular columnists appear in this month's issue. Rebecca Wirfs-Brock continues her Object-Oriented Design column by discussing the development phases of an object-oriented application. As Rebecca points out: "No matter how great the Smalltalk development environment, it isn't a replacement for planning, designing, and some amount of discipline." In Getting Real, Juanita Ewing concludes a two-part article on the appropriate use of class variables and class instance variables. Finally, in the GUI column, Greg Hendley and Eric Smith discuss the graphics model in Objectworks\Smalltalk Release 4.

Also in this issue, Martin Osborne and Ann Cotton review Dave Smith's new book entitled *Concepts of Object-Oriented Programming*. Despite the title, this book is entirely Smalltalk oriented and provides an introduction to the concepts underlying both OOP and Smalltalk. The review looks at the book from both the perspective of an experienced OOP person and of a novice.

Our publisher tells us that subscriptions to *The Smalltalk Report* are continuing to grow at a healthy rate and that an unusually high number of our readers are from Europe. A special welcome to you—we hope you enjoy the newsletter and look forward to meeting you in February at OOP '92 in Munich, Germany, or LOOK '92 in Copenhagen, Denmark.

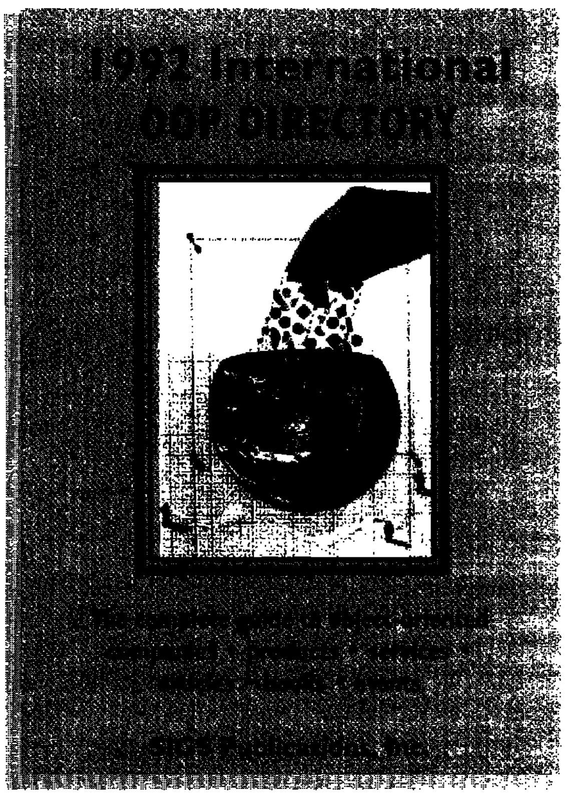
Finally, a big thank you to everyone who has helped us take *The Smalltalk Report* from an idea to fruition in 1991, and an invitation to all of you in the Smalltalk community to contribute to the report in '92.

[illegible] [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 COOT, Inc., a member of the SIGS Publications Group, 588 Broadway, New York, NY 10012 (212)274-0640. © Copyright 1992 by COOT, 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 1992 INTERNATIONAL OOP DIRECTORY

This is a limited edition offer



If you're using object technology, you should have a copy of the International OOP Directory.

Now You Can Have Access To Accurate, Up-To-Date And Complete Information On The Entire O-O Technology Market Right At Your Fingertips

This handy 425-page sourcebook is packed with everything you need to make OOP-related purchasing decisions:

- *Over 350 companies*
- *Nearly 500 products*
- *Consultants & services*
- *Reprints of landmark articles*
- *Bibliography by author/article*
- *Conferences & seminars*
- *Cross-referenced by languages and systems supported*

The International OOP Directory

❑ Send me ____ copy/copies at $69.00 each.
• Add $4 for U.S. shipping/handling
• Add $12 foreign airmail/handling
Total: $____

PAYMENT METHOD:
❑ Check enclosed (payable to *OOP Directory*).
Foreign orders must be prepaid in US dollars and drawn on a US bank.
❑ Charge my ❑ Visa ❑ MC

Card # ____

Exp. Date ____

Signature ____

Name ____

Company ____

Address ____

City ____

State ____

Country ____

Zip ____

Phone ____

Return to: **The International OOP Directory**, Subscriber Services, P.O. Box 3000, Dept. DIR, Denville, NJ 07834, or order by phone (212) 274-0640 or FAX (212)274-0646

continued from page 1...

Class-based ownership has the advantage that clear responsibility is established at a level of granularity that reflects the belief that objects are primary. By owning a class, a developer can ensure that the class is internally consistent and is clean.

Function-based ownership has the advantage of more closely reflecting how work proceeds in software development: Functions (from the users' point of view) are added, changed, and removed. In general, this means that the work of individuals will be independent.

The weakness of each of these approaches is the strength of the other. Since adding functionality tends to be distributed over a number of classes, using class-based ownership can slow down development as various individuals need to implement methods to complete some function. Likewise, function-based development can easily result in poor quality classes since no single individual is working to keep the class clean.

INCREMENTAL DEVELOPMENT

As illustrated above, using Smalltalk encourages an incremental style of software development. This style is demonstrated by the individual developers and is also demonstrated by teams of Smalltalk programmers when they are supported by management in this approach. The change management system used in projects is a key element in the successful adoption of this style.

When incremental development is used in a project, it also has an effect on the testing of the product. Testing can begin much earlier than with classical development as releases can be prepared with subsets of the full functionality. This means that testing can proceed in parallel with the development. The change management system needs to support this practice.

GOALS

Taking all the above into account results in the following goals for a change management system to support Smalltalk development:

1. Support incremental development.
2. Support the "natural" style that good Smalltalk programmers use.
3. Support the idea of ownership.
4. Support the timely release of versions.

For the purposes of this discussion, we assume that a change management system accepts changes from individual programmers, validates changes against the ownership rules, and periodically produces a "build" using the validated changes. A system tightly integrated with the development environment would prohibit the developer from making any changes that are not valid.

PROPOSAL

The key to our proposal is that the change management system operate at all times in one of two phases: the expansion phase or the consolidation phase. In each phase, the ownership of classes and methods is different.

In the expansion phase, ownership is function based. This allows individuals to focus their activity on the functions that they have been assigned. When programmers receive a new build in this phase, they will be able to continue their development as if they were just using their image from the prior build, that is, the image should always work.

In the consolidation phase, ownership is class based. This phase provides time for clean up of existing functionality, no new functionality should be added. There is no guarantee that the image will work for all functions after a build.

In a typical project, these two phases would alternate. Expansion is used when adding functionality, and consolidation is used when cleaning up. The relative proportions of the phases will change over the duration of the project. At the start, the activity is primarily expansion since the bulk of the new functionality is added. In the later stages, the dominant phase is consolidation as code is tuned, code bulk is reduced, and classes are cleaned for reuse.

When developing in Smalltalk, the time spent in the two phases is relatively short. Assuming that a relatively complete analysis has been performed, it appears that in the initial stages of a project the expansion phases should be no longer than three or four weeks. Hence, functionality needs to be broken down to support these time frames. This is most easily done by dividing functions breadth first in the early stages.

SPECIFIC RULES FOR THE PHASES

Below, I discuss some guidelines for the activities allowed in each phase.

EXPANSION PHASE

During the expansion phase, classes and methods are owned by functions. This ownership should be retained until the beginning of a consolidation phase. No more than one individual should be assigned to a function, and not all classes need to be owned.

Rules:

1. The creation of a class establishes ownership of the class.
2. The changing of a class definition establishes ownership of the class.
3. Methods added to nonowned classes establishes ownership of the method.
4. Methods that are not owned cannot be removed or changed.
5. Methods cannot be added that will change the method look-up of existing messages.

These rules are designed to meet two objectives:

1. Allow individual developers to continue their development between builds as if they were working in their own image.
2. Minimize the conflict between different developers at build time.

The rules as presented are not sufficient. Two developers may add the same class or method to an existing build and create a conflict. In practice, this would be minor, and the validation system would catch it.

Some may consider the requirement of only one person working on a function to be excessively restrictive when the function is large. In this case, we suggest that a single individual develop the initial functionality in a sketch form, providing the initial classes and minimal methods. The ownership of these can then be factored into smaller functions for individual assignment.

CONSOLIDATION PHASE

All classes should be owned by individuals during this phase. Individuals clearly will need to own more than one class. Use this rule: Changes can only be made by owners.

The objectives of this phase are to improve the quality of the classes while retaining the functionality at the beginning of the phase. No additional functions should be added.

Since message selectors can be changed and removed, builds may not work for all functions. To solve this, first add the new methods to a build, publish the replacement selectors to the owners of the senders of obsolete messages, and then remove the old methods when they are no longer used. This approach will ease most consolidation efforts.

MANAGEMENT IMPLICATIONS

The alternating phases of this proposal strongly support incremental development. They allow projects to be grown by adding functions in small groups. This has two positive effects: The progress of development can be more easily measured, and testing can begin relatively early in the development. The consolidation phases provide space for developers to reflect on their work and to think about the reuse of developed classes. Management has some flexibility in scheduling by reducing the consolidation time at the risk of having less reusable and suitable code.

A prototypical project plan might look like Table 1 for a set of function groups {A, B, C, and D}. Each row holds the parallel activities of the development and testing groups.

From Table 1, it is easy to see how better control of the project is possible and also how consolidation phases could be adjusted to alter the project schedule.

Managers need to understand, with object-oriented development consolidation is a natural phase and needs to be supported. When it is, systems are developed that are more reliable and provide greater reuse for future development.

Table 1. A prototypical project plan.

Development	Testing
e(A), deliver A	
c(A), deliver A'	A
e(B), deliver B	A'
c(B), deliver B'	B, regression test on A
e(C), deliver C	B'
c(C), deliver C'	C, regression test on A, B
e(D), deliver D	C'
c(D), deliver D'	D, regression test on A, B, C
deliver final	D'
	regression test on A, B, C, D

SUMMARY

I have proposed a change management system that is based on our observations of seasoned Smalltalk developers at work and is designed to offer management more control in development. By imitating the natural workings of good developers, such a system will result in software that is more suitable, timely, and provides greater reuse potential. ■

Jeff McKenna has been involved in the software industry since 1963 and object-oriented technology since 1981. He is the founder of McKenna Consulting Group, which offers services in object-oriented technology. He has been actively involved with OOPSLA, the premier object-oriented conference. Jeff is well known as a speaker, having introduced O-O concepts at various convferences throughout the world. He was the founding editor of the Hotline on Object-Oriented Technology, *an industry newsletter.*

A CASE IN POINT

Charles-A. Rovira

In his keynote speech at Developers' Conference '91, the first annual Digitalk and *BYTE Magazine* Smalltalk/V developers conference, Daniel H. H. Ingalls remarked that the reason Smalltalk didn't have a **case** statement was as a result of a conscious decision by the Smalltalk development group to keep the syntax as simple as possible. Smalltalk had enough of a learning curve to round without syntactic sugar cluttering up the language specification. However, Dan did confide that it would have been trivial to add a **case** statement to the compiler. A case statement is essentially a series of if-then constructs. The team at PARC felt that programmers might as well describe them this way and, as far as Smalltalk went, they left well enough alone.

WHAT IS THE TRUE "MOTHER OF INVENTION?"

While the PARC crew might have felt content to let sleeping dogs lie, I, being an indolent person who gets lost in '[' ']' pairs and who hates writing all those **ifTrue: ifFalse:** and **and: or:** expressions, quickly decided to remedy the situation without undue resort to the compiler. The remedy I concocted had to eschew the compiler because, unlike Smalltalk-80, Digitalk attempts to hide its compiler behind classes with no real names and methods without source code. Given the availability of clever hacks, as well as hackers, I can report that though the approach will yield to a concerted effort, it is sufficient to discourage the casual Smalltalk user, should such a beast ever be found. Since necessity is the mother of invention and sloth is the mother of necessity, here is the case statement I cooked up over a weekend.

Listing 1 includes all of the code required to add case statement capability to Smalltalk. Listing 2 is a small sample that uses the case statement to simplify its coding.

Listing 1. The case statement components.

```
Association subclass: #Case
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''

Case methods
, aCase
        "Answer a collection of cases"
    ^ Cases with: self with: aCase


OrderedCollection subclass: #Cases
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''

Cases methods
, aCase
        "Answer an OrderedCollection containing all the
            elements of the receiver followed by the new case"
    ^ self copy
        add: aCase;
        yourself

Object methods
case: aCollection
        "Handle parameterless cases"
    ^self case: aCollection test: nil eval: nil

case: aCollection parm: anObject
        "Handle single parameter cases"
    ^self case: aCollection test: anObject eval: anObject

case: aCollection test: test eval: eval
        "Answer the result of the first case that rings true"
    | trueCase conditionBlock actionBlock |
    (aCollection isKindOf: Cases)
        ifFalse: [^self error: 'case statement improperly structured'].
    trueCase := aCollection
        detect: [:each |
            (each isKindOf: Case)
                ifFalse: [^self error: 'case improperly structured'].
            ((conditionBlock := each key) isKindOf: Context)
                ifFalse: [^self error: 'case discriminant
                    improperly structured'].
            (conditionBlock blockArgumentCount = 1)
                ifTrue: [conditionBlock value: self]
                ifFalse: [(conditionBlock blockArgumentCount = 2)
                    ifTrue: [conditionBlock value: self value: test]
                    ifFalse: [conditionBlock value]]]
        ifNone: [^nil].
    ((actionBlock := trueCase value) isKindOf: Context)
        ifFalse: [^self error: 'case imperative
            improperly structured'].
    (actionBlock blockArgumentCount = 1)
        ifTrue: [^ actionBlock value: self]
        ifFalse: [(actionBlock blockArgumentCount = 2)
            ifTrue: [^ actionBlock value: self value: eval]
            ifFalse: [^ actionBlock value]]
cases: aCollection
        "Handle parameterless cases"
    ^self cases: aCollection test: nil eval: nil
```

Listing 1., cont.

```
cases: aCollection parm: anObject
        "Handle single parameter cases"
    ^self cases: aCollection test: anObject eval: anObject

cases: aCollection test: test eval: eval
        "Perform all cases that ring true"
    | trueCases conditionBlock actionBlock |
    (aCollection isKindOf: Cases)
        ifFalse: [^self error: 'cases statement improperly structured'].
    trueCase := aCollection
        select: [:each |
            (each isKindOf: Case)
                ifFalse: [^self error: 'cases improperly structured'].
            ((conditionBlock := each key) isKindOf: Context)
                ifFalse: [^self error: 'cases discriminant
                    improperly structured'].
            (conditionBlock blockArgumentCount = 1)
                ifTrue: [conditionBlock value: self]
                ifFalse: [(conditionBlock blockArgumentCount = 2)
                    ifTrue: [conditionBlock value: self value: test]
                    ifFalse: [conditionBlock value]]].
    trueCases do: [ :each |
        ((actionBlock := each value) isKindOf: Context)
            ifFalse: [^self error: 'cases imperative
                improperly structured'].
        (actionBlock blockArgumentCount = 1)
            ifTrue: [^ actionBlock value: self]
            ifFalse: [(actionBlock blockArgumentCount = 2)
                ifTrue: [^ actionBlock value: self value: eval]
                ifFalse: [^ actionBlock value]]

Context methods
case: aBlock
        "Answer a 'sentive' case"
    (aBlock isKindOf: Context)
        ifFalse: [^super case: aBlock].
    ^ Case key: self value: aBlock

blockArgumentCount
        "Answer the number of arguments for the block. Note that
            in /V PM, the isMemberOf: aBlockClass message would be
            used instead; e.g., (aBlock isMemberOf: TwoArgumentBlock),
             (aBlock isMemberOf: OneArgumentBlock)."
    ^ blockArgumentCount

UndefinedObject methods
case: aBlock
        "Answer an 'insentive' case"
    ^ [true] case: aBlock
```

Listing 2. A sample of case statement use.

```
Number methods
printAs: aFormat
        "Append the ASCII representation of the receiver to aStream.
        Filter through aFormat.
        Test: 123.45 printAs:'$,#$$.00 '
        Test it with other numbers and other format strings."
    | temp answer sign format digit |
    temp := (Pattern new: '.') match: aFormat index: 1.
    temp isNil
        ifTrue: [temp := self abs]
        ifFalse: [temp := (self abs) *
            (10 raisedTo: ((aFormat copyFrom: temp x to:
                aFormat size) select: [:ch | #( $# $0 $$ ) includes:
                ch ]) size).
            temp := temp + 1 truncated].
    answer := String new: aFormat size.
    sign := self negative.
    format := aFormat reversed.
    answer := format collect: [:formatCharacter |
        (formatCharacter case:
            (([:x | x = $#] case:
                [digit := Character value: (48 + ((temp rem: 10)
                    truncated)).
                temp := temp // 10.
                (digit = $0 and: [temp = 0]) ifTrue: [$ ] ifFalse:
                    [digit]]) ,
            ([:x | x = $0] case:
                [digit := Character value: (48 + ((temp rem: 10)
                    truncated)).
                temp := temp // 10.
                digit]) ,
            ([:x | x = $$] case:
                [digit := Character value: (48 + ((temp rem: 10)
                    truncated)).
                temp := temp // 10.
                (digit = $0 and: [temp = 0]) ifTrue: [$$] ifFalse:
                    [digit]]) ,
            ([:x | x = $, ] case:
                [temp = 0 ifTrue: [$ ] ifFalse: [$,]]) ,
            ([:x | '-+{ }()DBCRdbcr' includes: x] case:
                [:x | (sign ifTrue: ['—  ()DBCR    '] ifFalse:
                    [' +()      DBCR'])
                    at: ('-+{ }()DBCRdbcr' indexOf: x ifAbsent:
                    [self halt])]) ,
            (nil case:
                [:x | x])))].
    ^answer reversed
```

Basically, all that was required was to examine how a case statement really works. A case statement associates conditions with actions. (In the search for more highfalutin bureaucrateese and marginally more accurate bafflegab, I have usurped the terms discriminant and imperative.) There are times when only a single action need be taken, when only a single case need be invoked, and there are times when all the conditions should be evaluated and all suitable cases need to be invoked.

This "simple" English version of the requirement definition leads to the following class definitions:

- a Case that associates a discriminant with an imperative and should therefore logically be a subclass of Association
- a Case that lists all individual cases for evaluation and should therefore logically be a subclass of OrderedCollection

In addition, proper support must be found for the evaluations both of the conditions and of the associated actions and for the invocation of the evaluation of the conditions.

The fundamental idea behind the implementation of the case statement was to use the deferred evaluation and late binding facilities inherent in Smalltalk contexts, a.k.a. blocks, to leave the actual evaluation of the discriminant of a case and the execution of the imperative of a case whose discriminant evaluates to true until it is truly required. The principle is the same as the passing of blocks of code to the **detect:** and **select:** instance methods of the abstract class **Collection**. In fact, these very methods are what allow the detection or selection of none, one, or many cases as per their discriminants into a temporary collection of cases whose imperatives have to be evaluated.

“There are times when...only a single case need be invoked, and there are times when all of the conditions should be evaluated and all suitable cases need to be invoked.”

The case statement can be made without any parameters. This means that the block evaluating the case must be entirely dependent on external data with either a single parameter to the block (which means that the case can be based on the object itself) or with two parameters (which means that the case can be based on an object for the evaluation of the case while being based on another object for the evaluation of the result).

The sad part is that I can't for the life of me remember what could have possibly led me to write the methods with two parameters. There was a very good reason at the time but it is lost in the mists of time and in some old client's /V PM code.

JUST IN CASE

As Listing 1 reveals, the code for both **Case** and **Cases** is simple. It consists of the polymorphic implementation of the single method: “,”. (Of course this begs the question, “Is the method really a single method if it needs to be in two places at once?”) The rest of the code presented is concerned with usability.

The other methods are broken down into three groups:

- **UndefinedObject>>case:**, which is included for ease of use
- **Context>>case:**, which will either build a single case or cause a case statement to be evaluated, and **Context>>blockArgumentCount**, which will supply the number of arguments for a particular case evaluation block
- **Object** methods, which will cause any object to be used as the point of origin during the evaluation of a case statement. The methods are divided into two types:
    - **case** methods that will only look until the first successful evaluation
    - **cases** methods that will execute the actions of all conditions that evaluate to true

The statements are also divided into three flavors:

- without parameters—the evaluation of the conditional block will determine the truth of the assertion
- with a single parameter—the evaluation of the conditional block on the object itself will determine the truth of the assertion
- with two parameters

THE CASE FOR THE PROSECUTION

Listing 2 is a “quick and dirty” method for printing numbers filtered through a format. The idea behind this method was to provide something similar to the **PRINTUSING()** statement found in BASIC. It is a simple way to format numbers in a report using a print line layout “painted” by a CASE tool.

The method inverts the format string and pastes the value of the receiver one digit at a time. That part is standard. What is different is the scanning of the format string to look for somewhere to paste in the digit. This routine relies on a loop to examine each character of the format string and uses a **case:** statement to decide what to do with the character.

The routine is not very sophisticated and doesn't handle format string overflow. However, it will put plus or minus signs, parentheses, or debit/credit annotations depending on the sign of the number. The format selected as its space, dollar sign, or zero fills the number and handles leading commas. It is heavily dependent on the intelligence of the programmer and does not check if the formatted string makes any kind of sense. (This is actually an advantage as the routine can be used for nonstandard formatting needs.)

The heart of the routine is a loop that sweeps the format string, character by character, and relies on a case statement to act according to the state of the character. The state of the receiver (remember the number being formatted?) is totally irrelevant to the formatting process.

The code uses Digitalk's Smalltalk/V Mac 1.2 and System 7.0 on a Macintosh IIcx.■

Now based in Ottowa, Canada, Charles-A. Rovira has been involved with data processing since 1975 and with Smalltalk and other object-oriented technologies since 1987. His CompuServe ID is [71230,1217]. He'll admit to some unusual literary influences such as Douglas Adams, Terry Pratchett, and D. H. Lawrence. Also, Kierkegard, but why bring him up?

Rebecca Wirfs-Brock

The phases of an object-oriented application

There is never enough time to get it absolutely, perfectly right. I was lured to computer science by the fact that I could spend hours and hours working on elegant solutions to fairly simple problems. Often, when my code became too difficult to follow, I could find a simpler design if I had the courage to back up and rethink my almost workable solution. Things actually got better if I relaxed and did not try so hard to force my program to work. After I got my degree and an engineering job, I found that not only did my code have to work, I had to provide a detailed plan for my work and estimate the completion date of each major task. Assignments no longer could be easily completed within a week. As a consequence, I learned how to subdivide a large problem into smaller, more manageable activities. I also learned to pad my estimates (to account for the unaccountable) and to reassess my plan whenever I achieved a subgoal.

Object-oriented technology can add complexity to the software development puzzle. Object-oriented design techniques and programming languages provide good tools for handling abstractions and developing potentially reusable software. Yet, what is the additional cost of developing reusable code? It is hard enough to plan and deliver software on time, within budget, and meeting customer expectations with traditional development methods. Designing and implementing for reuse presents a totally new set of challenges.

A class that has been designed and implemented to be used in more than one application probably requires more effort than a class designed to work within a single application. However, shouldn't all classes be designed to be understood and usable by other programmers, regardless of their general utility? Certainly, not all classes are worth equal time and attention. Since time is limited, what is an appropriate way to divide the time spent developing various parts of an application? The challenge is to know when and where to apply extra effort. It's also important to know when to stop tweaking code for the sake of "making it better" when returns will be meager.

TYPICAL APPLICATION STRUCTURE

An object-oriented application of even moderate complexity is naturally decomposed into several major *subsystems*. Each subsystem consists of objects from classes that share the overall workload of the subsystem and collaborate to get the subsystem's tasks accomplished. In a well-factored design, objects within each subsystem primarily collaborate with each other. Certain key objects handle requests from other objects outside the subsystem. In general, however, few objects within a subsystem are visible outside the subsystem.

In many designs, there also are a number of general utility classes. Smalltalk environments provide a comprehensive set of container, graphics, and user interface classes. In addition to this valuable class library, many applications add their own specific utility classes. Rather than having each subsystem design consist of its unique but perhaps only slightly different classes, a common class library is developed and used throughout the entire application. These classes serve to enforce common error handling policies, support default behaviors, or encapsulate information passed between subsystems.

“It is hard enough to plan and deliver software on time, within budget, ... [etc.]. Designing and implementing for reuse presents a totally new set of challenges.”

A DEVELOPMENT TIME LINE

The overall development process can be roughly divided into distinct phases. The first stage of any design consists of exploring possible alternatives. Major subsystem partitioning strategies are determined. An initial model of the key design objects is proposed. Once this initial model has been developed, efforts shift into a detailing phase where precision is added to initial decisions. Subsystems and the classes within them are sufficiently elaborated and then implemented.

Each subsystem will progress at a different pace due to variations in complexity and according to the abilities and experiences of its designers. However, any subsystem will pass through most of these steps:

1. **Specification.** During this stage, a rough idea of the purpose of the subsystem and the services it will provide is

proposed. An estimate of the subsystem's complexity can be made. This estimate may include a list of key classes (perhaps including their names and a brief description) and some measure of their complexity and projected general utility as well as an estimate for the time required to complete an exploratory design.

2. **Exploratory design.** During this stage, key objects and their interactions are modeled. An initial pass is made at defining each key class' role and responsibilities. Several additional layers of each subsystem design can be elaborated. Services available to objects outside the subsystem are specified in greater detail. Assumptions about services provided by other objects and subsystems are proposed. These assumptions will need review and refinement in the context of the overall application architecture.

3. **Detailed modeling.** Elaboration of the initial exploratory design means extensive review and refinement of the initial model. Classes are scrutinized for appropriate factoring of responsibilities. A lot of time can be spent making slight readjustments of object roles and responsibilities to minimize interobject dependencies and simplify the design. New supporting classes may be created to further reduce coupling between classes. Permissible patterns of collaboration between objects can be formalized through contracts that spell out services used by specific clients. Finally, class inheritance hierarchies can be developed. Common responsibilities can be found, and superclasses can be created that generalize behavior common among several classes.

4. **Implementation.** Whether one calls finalizing internal details of each class the last step in detailed modeling or the first task of implementation isn't important. However, at this point a number of design issues that have been deferred must now be decided. Decisions must be made about the representation of each class' attributes or characteristic properties. The choices are to derive an attribute from other information or to store it as an instance variable. New classes may be constructed to model attributes if existing classes aren't appropriate. Operations must be decomposed into reasonable substeps and implemented. Careful attention must be paid to ensure consistent, clear message protocols. The fine details of abstract classes must be developed and will be proven by the ease with which their subclasses can be implemented.

5. **Integration.** Another crucial point in any large application comes when subsystems developed in relative isolation (after agreeing upon basic intersubsystem interactions and publicly available services) are made to work together. Test stub methods and objects are replaced by their application stand-ins. It is at this stage that hidden assumptions about services provided and/or expected patterns of usage are uncovered and, once again, might need readjusting.

6. **Validation.** Once parts of the application are functioning, the operation of classes and subsystems can undergo extensive validation. It is reasonable to test a class in isolation (by developing test methods, adjusting its encapsulated state, and testing how it responds). It is also necessary to validate the overall behavior of major subsystems in the actual working environment.

7. **Cleanup.** Once a subsystem has been implemented and validated, it often merits further attention. A relatively minor sweep through the classes and working code can provide dramatic improvements in performance, code clarity, and robustness. The goal of this phase is to provide for better use and improved maintenance.

8. **Generalization for broader utility.** Once a subsystem is implemented and works well, its general utility can sometimes be improved. This activity needs to be carefully planned. Not all subsystems are significant enough or have enough potential utility to merit this extra effort.

There is a separate time line for each subsystem under development. Several major integration points can add subsystem functionality in varying stages of maturity. Figure 1 shows a time line for a hypothetical object-oriented application. The time line for utility class development is intentionally missing. In an ideal situation, utility classes would be developed along with the subsystems that use them. They would need refining throughout the project. In this hypothetical application, developers of one subsystem skipped over detailed modeling and launched right into implementation. This might have been due to an overeager implementation team or because the subsystem was simple enough to not warrant much detailing. Many subsystems were modeled in detail and passed through most steps. However, only one subsystem is shown being generalized for even broader utility. Most subsystems (at least during this time line) never were generalized.

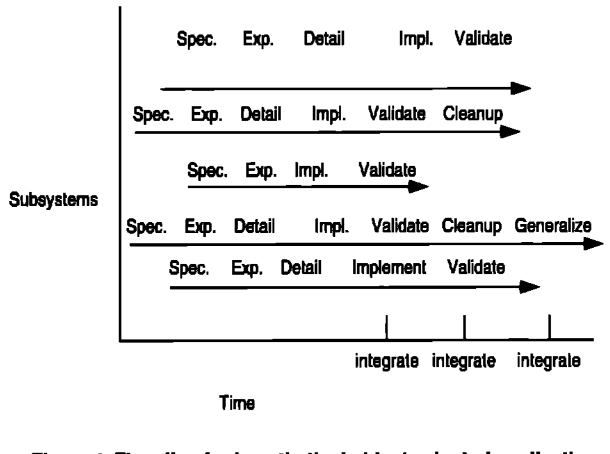
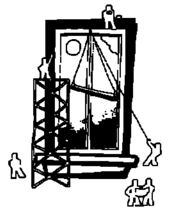

Spec. Exp. Detail Impl. Validate
Spec. Exp. Detail Impl. Validate Cleanup
Spec. Exp. Impl. Validate
Subsystems
Spec. Exp. Detail Impl. Validate Cleanup Generalize
Spec. Exp. Detail Implement Validate
integrate integrate integrate
Time

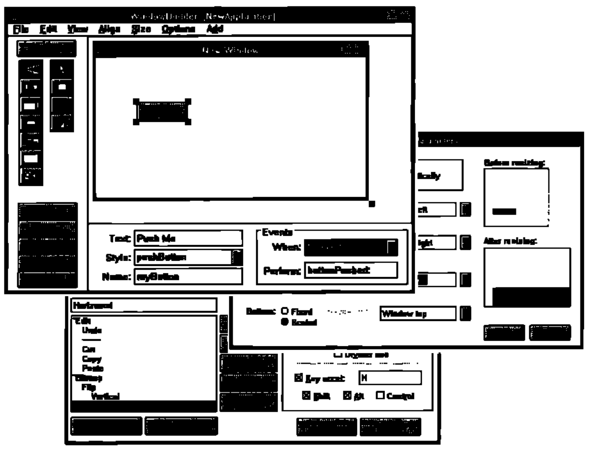
Figure 1. Time line for hypothetical object-oriented application.

COOPER + PETERS

WINDOWBUILDER



The Interface Builder for Smalltalk/V



Text: Push Me
Style: pushButton
Name: myButton
Events
When:
Perform: buttonPushed

"... 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

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.

Until March 31st, WindowBuilder/V PM will be available at an introductory price of $295, $100 off the list price of $395. WindowBuilder/V Windows sells for $149.95. Both include an unconditional 60 day guarantee.

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

WHERE TO SPEND TIME AND EFFORT

Obviously, all classes are not of equal value or worth,.and many classes in object-oriented applications are developed for use, not reuse. But, to be used (by anyone other than the original author) or enhanced in future maintenance releases, classes still need to be engineered and implemented with care. If an inadequate amount of time is spent in detailed modeling, implementation and maintenance costs can skyrocket.

One reasonable estimate I've applied to scheduling is that detailed modeling can take roughly twice as much time as initial exploration. This estimate was based on the assumption that the designers had a good working knowledge of the problem area and weren't trying to learn about the application requirements as well as object technology. If the team has been fairly disciplined about detailing the design model, then implementation time can be shortened.

If the design team is relatively new to both the application and object technology, it may be tempting to move directly from an exploratory model right into implementation. This may be a reasonable strategy to get the team thinking and implementing in objects. However, resist the urge to bolt directly to implementation. Spend some time reviewing the initial model. Try to assess high-leverage areas that are worth extra design time as well as areas where the design still seems unclear. Given an inexperienced team, the initial implementation may well turn out to be a prototype. The application will more than likely need to be redesigned and reimplemented following a more disciplined approach once the basic model and application objectives are understood.

“ Perhaps only 20% of the application classes are worth spending 80% of the total time devoted to reuse improvements. ”

Clearly state goals for the overall quality level expected for each class and subsystem. Establish targets for each subsystem for the amount of refinement and generalization warranted. Perhaps only 20% of the application classes are worth spending 80% of the total time devoted to reuse improvements. As work progresses on each subsystem, stop and reassess progress shortly before and after each major milestone. Examine the flaws and issues that have been uncovered. Glossing over serious gaps in design or implementation will only delay later consequences when the cost of backtracking and fixing are higher.

It requires discipline on the part of management and the design team to pause to measure progress and quality and to plan for the next phase. Object-oriented software development should not be an excuse for throwing out proven development practices, even if the tools and techniques are a big improvement. Here are some characteristics of a reasonably well-thought-out design:

- Classes have been factored to do one thing well. Each class has a singular, clearly stated purpose and the implementation follows the design intent. The alternative is fewer classes that do several more things adequately.
- Public interfaces to classes are straightforward and simple to understand. In general, messages don't have lots of arguments. It's even better if using an object doesn't require understanding complex modes, switches, or a complicated internal state machine.
- Methods have been decomposed into a several discrete steps. These steps are implemented by sending messages to the receiver (self) or delegating tasks to objects referenced through instance variables. The alternative is lengthy, long-winded methods.
- There are a number of classes having roles of manager, coordinator, or information repository. They provide generally useful services that are straightforward and readily understood. These classes provide useful mechanisms, infrastructures, and the "glue" for the rest of the system, reducing the overall complexity of many other classes.
- Class inheritance hierarchies may have been developed. There may be abstract classes at the root of these hierarchies. The purpose of developing class hierarchies with abstract classes is to abstractly specify behavior common to a number of existing subclasses. The alternative is rather flat inheritance hierarchies with little or no commonly shared behavior. Future additions, extensions, and modifications will be easier to make if time has been spent building clean, understandable class hierarchies.

ENHANCING REUSE AND REDUCING MAINTENANCE COSTS

Refining classes for reuse is analogous to optimizing code for improved performance; neither happen by chance, but well-planned and executed improvements can be quite dramatic. Here are some ways to improve existing classes and subsystems:

- Isolate replaceable features and decompose algorithms into subparts (which can be overridden by new subclasses).
- Encapsulate instance variables. Rewrite class code to call accessing methods. This allows subclasses to change and/or augment inherited instance variables without having to rewrite superclass code.
- Spend time streamlining collaborations between subsystems. Reduce the number of classes that are visible outside the subsystem.
- Augment classes that worked adequately for one application to increase their utility. Rework class hierarchies and create both abstract classes to represent useful generalizations and new subclasses that represent useful specializations.
- Improve the legibility and understandability of existing classes. Simplify message protocols and make them more consistent. Augment class and subsystem documentation with discussions on intended usage, sample code, and calling sequences. Add typical calling sequences to existing code as comments.

No matter how great the Smalltalk development environment, it isn't a replacement for planning, designing, and some amount of discipline. Developing an object-oriented application involves new ways of thinking and structuring solutions. The biggest payoff comes when sound engineering practices are added to the development picture. ■

SUGGESTED READING

1. Moore, J. M., and S.C. Bailin. Domain analysis: framework for reuse, in *Domain Analysis and Software Systems Modeling*, Ruben Prieto-Diaz and Guillermo Arango, eds., IEEE Computer Society Press, 1991, pp. 179 – 203.
2. Wirfs-Brock, A., and B. Wilkerson. Variables limit reusability, *Journal of Object-Oriented Programming* 2(1): 34 – 40, 1990.

Rebecca Wirfs-Brock is the Director of Object Technology Services at Instantiations and coauthor of Designing Object-Oriented Software. *She is the program chair for* OOPSLA '92. *She has sixteen years of experience designing, implementing, and managing software products. During the last seven years she has focused on object-oriented software. She managed the development of Tektronix Color Smalltalk and has been immersed in developing, teaching, and lecturing on object-oriented software. Comments, further insights, or wild speculations are greatly appreciated by the author, who can be reached via email at rebecca@instance.com or by mail at Instantiations, 921 S.W. Washington, Ste. 312, Portland, OR 97205.*

Juanita Ewing

How to use class variables and class instance variables

In last month's column, we discussed how classes that use class variables can be made easily reusable with a few coding conventions that make it easier to create subclasses. However, class variables are shared by a class and its subclasses. Often, this is inappropriate, and a subclass needs to override inherited data. A better implementation choice for a particular problem is often a class instance variable rather than a class variable.

WHAT ARE CLASS INSTANCE VARIABLES?

Class instance variables are those that belong to a class. Smalltalk systems rely on this facility. For example, each class stores its name in a class instance variable. Just as each instance has its own values for instance variables, each class has its own values for class instance variables. Unlike class variables, these variables are not shared by all instances of a class.

Only class methods can reference class instance variables. Direct references to these variables are not allowed from instance methods. Instance methods that need the information stored in a class instance variable must send a message to a class method, which can return the requested information.

Class instance variables, but not their values, are inherited. Since each class has its own values for class instance variables, there is no sharing between a class and its subclasses.

In last month's issue, we discussed how the convention of using get and set methods for class variables, rather than direct references, made it easier to create subclasses by minimizing the number of methods that must be overridden. For example, in Figure 1 this made the class ListInterface more reusable.

“A common mistake is to use class variables in places where sharing between a class and its subclasses is inappropriate.”

However, a problem remained with the class ListInterface. Another class variable was created to provide a different menu resulting in CalculatedListInterface having two class variables, one of which (ListMenu) is not used. The problem is that class variables share the data between subclass and superclass. However, we can avoid the sharing of the class variable ListMenu through the use of class instance variables.

This version of ListInterface, illustrated in Figure 2, defines its menu with a class instance variable. The class

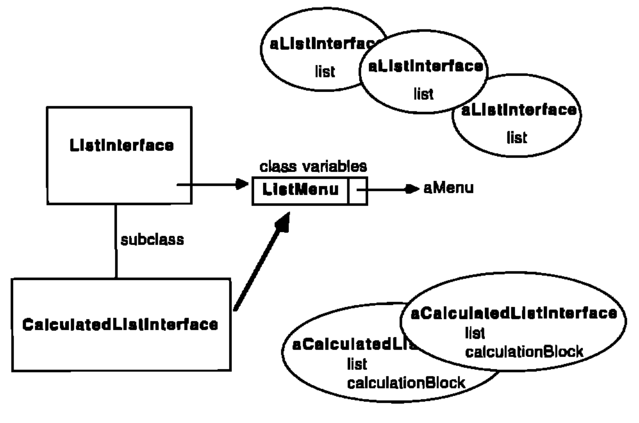


Figure 1. Coding conventions increase the reusability of classes implemented with class variables.

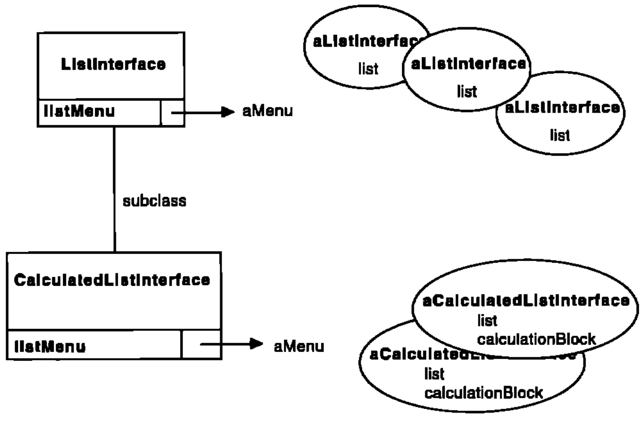


Figure 2. Subclasses have their own copy of class instance variables.

silence...

the end to your Smalltalk/V troubles

- full multi-user project management
- source code version control
- automatic change documenting
- release packaging
- source code hiding
- code performance profiling
- change log browser and restorer
- installer with global renaming capability

introductory pricing
until March 31st, 1992

$99.95 Windows version available immediately
OS/2 and Mac versions - CALL!
source code included

digamma solutions

Unit 6, 387 Spadina Avenue, Toronto, Ontario, Canada, M5T 2G6
Phone: (416) 351-6833 Fax: (416) 408-2850

methods in **ListInterface** directly reference the class instance variable. Instance methods cannot directly reference **listMenu**, but instead send messages to the class to access the value of **listMenu**:

```
ListInterface class
    initialize
        "Create a menu."

        listMenu := Menu labels: #('add' 'remove')

    menu
        "Return the menu."

        ^listMenu

ListInterface
    hasMenu
        "Return true if a menu is defined."

        ^self class menu notNil

    performMenuActivity
        "Perform the mouse-based activity for my view."

        self hasMenu
            ifTrue:[^self class menu startUp].
```

Now let's create a version of **CalculatedListInterface** that has a different menu. What does the developer need to do? The developer does not need to define a new variable. Each class has its own copy of the class instance variable **listMenu**. Class methods in **CalculatedListInterface** simply need to assign the appropriate menu to the class instance variable. How many methods need to be overridden? Only one:

```
CalculatedListInterface class
    initialize
        "Create a menu for calculated lists."

        listMenu := Menu labels: #('add' 'remove' 'print')
```

CalculatedListInterface has its own copy of the menu stored in the class instance variable **listMenu**. All methods that access this class instance variable work properly in subclasses because they reference the menu stored in their own class. This version of **CalculatedListInterface** contains only one method and uses all its defined variables, unlike the previous version that contained a class variable from the superclass.

Most classes, especially those created as stand-alone abstractions, should use class instance variables so that new subclasses can be created with minimal effort. A common mistake is to use class variables in places where sharing between a class and its subclasses is inappropriate.

WHICH VERSION OF ListInterface IS MORE REUSABLE?

The version of the class **ListInterface** that implements the menu with a class instance variable is more reusable than the version that uses a class variable. Fewer methods need to be overridden to create a subclass with a different menu. The version implemented with class variables requires a new class variable, while the version implemented with class instance variables does not.

Class instance variables are an important part of Smalltalk because they provide an important mechanism by which more reusable classes are created. All Smalltalk dialects have class variables, but only Smalltalk-80-derived dialects contain class instance variable support as delivered by the vendor. However, Smalltalk/V can be extended to support user-defined class instance variables with just a handful of methods.

Whenever possible, serious developers of reusable Smalltalk code should use the coding conventions discussed in this article and class instance variables. Class variables should be used only when there is an explicit need for shared variables because they limit the reusability of classes. ■

Juanita Ewing is a senior staff member of Instantiations, Inc., a software engineering and consulting firm that specializes in developing and applying object-oriented software projects, and is an expert in the design and implementation of object-oriented applications, frameworks, and systems. In her previous position at Tektronix Inc., she was responsible for the development of class libraries for the first commercial-quality Smalltalk-80 system. Her professional activities include Workshop and Panel Chairs for the OOPSLA conference.

Greg Hendley and Eric Smith

Objectworks\Smalltalk Release 4: the graphics model

Objectworks\Smalltalk Release 4 (R4) by ParcPlace systems provides a platform-independent virtual image that promotes complete portability of Smalltalk applications between various host systems. This level of portability is obtained by isolating the virtual image from any information about how input is gathered from, or output sent to, the host system. In our last column, we explained how this trick is accomplished on the input side of the coin. In this column, we'll take a look at how the output side is handled.

PRIMARY CLASSES

There are two primary classes used in displaying information in R4: **DisplaySurface** and **GraphicsContext**. It takes an instance of each to conduct any sort of display operation. **GraphicsContext** is the active component. Instances of this class are sent messages to draw lines, set colors, fill rectangles, display text, and so forth.

DisplaySurface is an abstract class representing an object that can be drawn on. An instance of one of its subclasses, **ScheduledWindow**, for example, is required to conduct any drawing operations with a **GraphicsContext**. The **DisplaySurface** is a passive partner in these operations, simply providing information to the **GraphicsContext** so that it may conduct the operation properly.

This arrangement provides a clean interface between the virtual image and the host windowing system's drawing primitives. There is a single point of contact, **GraphicsContext**, regardless of the number and type of **DisplaySurfaces** used. This division of labor is like that seen in Digitalk's Smalltalk/V for Presentation Manager and Windows. In Smalltalk/V, the **GraphicsMedium** and **GraphicsTool** are the passive and active objects in V. At least in this respect, the two main varieties of Smalltalk are moving closer together.

DISPLAYSURFACES

DisplaySurfaces are of three basic types: **Windows**, **Pixmaps**, and **Masks**. Instances of all of these classes represent entities that are external to Smalltalk. For example, examining the class **Window**, we see that it holds very little information. Instances of this class merely front for an entity within the world of the host windowing system. It is this host windowing system object that contains most of the real knowledge about what it means to be a window under that system. All that the Smalltalk object needs to know is how to identify the host windowing system object so that it may be queried and manipulated when necessary.

Under MS-Windows 3.0, this identification is easily accomplished by having instances of **DisplaySurface** maintain a copy of the windows handle of the host windowing system object that they represent. An instance of **ScheduledWindow** has an instance variable that contains a Smalltalk representation of the host window's handle.

> "It is this host windowing system object that contains most of the real knowledge about what it means to be a window under that system."

GRAPHICSCONTEXTS

DisplaySurfaces know how to create **GraphicsContexts** on themselves. The resulting object is an instance of **GraphicsContext** that knows the medium, a **DisplaySurface**, on which it is to draw. The two objects necessary to conduct display operations now exist.

Display messages may then be sent to the **GraphicsContext** to either present information to the user or record it on a Pixmap. The great advantage provided by R4 is that, no matter what system you are running on, XWindow System, MS-Windows 3.0, Macintosh, etc., the protocol for displaying is the same.

This is accomplished simply and elegantly with a relatively small set of primitives that implement all of the operations done to a **GraphicsContext**. Each of these primitives is implemented in the virtual machine to call the necessary graphics functions to accomplish the behavior defined for that primitive. Some of these primitives are quite simple: merely converting parameters and passing the call along to a single host windowing system graphics call. Others, on some systems, will be quite complex: performing a number of calculations to convert R4's idea of how a particular operation is done into several calls to host window-

VOSS

Virtual Object Storage System for

Smalltalk/V

Seamless persistent object management with update transaction control directly in the Smalltalk language

- **Transparent access to Smalltalk objects on disk**
- **Transaction commit/rollback**
- **Access to individual elements of virtual collections and dictionaries**
- **Multi-key and multi-value virtual dictionaries with query by key range and set intersection**
- **Class restructure editor for renaming classes and adding or removing instance variables allows incremental application development**
- **Shared access to named virtual object spaces**
- **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 ($375 to end of February 1992) + $15 shipping.
VOSS/Windows $750 ($475 to end of February 1992) +$15 shipping.
Quantity discounts available. Visa, MasterCard and EuroCard accepted.
Logic Arts Ltd. 75 Hemingford Road, Cambridge, England, CB1 3BY
TEL: +44 223 212392 FAX: +44 223 245171

ing system functions. Since all of this is done in the virtual machine, the Smalltalk programmer never has to see it.

THE MECHANICS

What actually happens when a graphic operation on a GraphicsContext disappears into a primitive? Let's look at the case under MS-Windows; the story on other systems is likely to be analogous. First, it seems safe to assume that the Smalltalk operation is eventually broken down into one or more calls to the graphics and windowing functions in the MS-Windows libraries.

Those familiar with MS-Windows will realize that no graphics operations can be done without having a handle for a device context (HDC). The HDC identifies, within MS-Windows, a complete graphics state for drawing on a particular display medium, such as a printer or a window. Naturally, the first task for the primitive that implements a particular operation for **GraphicsContext** is to obtain an HDC for the medium of that **GraphicsContext**. An HDC can be created by using the handle stored in the **GraphicsContext**'s medium.

If the HDC had to be created from scratch, as opposed to cached, it would then have to be programmed with the parts of the state of the **GraphicsContext** that are relevant to the current drawing operation. If the operation involves drawing lines, then the line width of the HDC must match that of the **GraphicsContext**. After the HDC is brought into compliance with the **GraphicsContext**, the MS-Windows calls that do the actual drawing would be called.

A CLEAN SEPARATION

In our last column, we explained the separation between the host system-specific aspects of input handling and the virtual image. Like the handling of information display covered above, by the time input gets to code in the virtual image, it has been normalized to a standard representation that is the same no matter what platform Smalltalk is running on. With the separation between the Smalltalk developer and the host windowing system's display mechanisms, the independence of the virtual image from the host environment is complete. This model allows the Smalltalk developer to work in the same image no matter what system he or she is working on. It also allows a particular virtual image to be moved from one kind of machine to another and to run without modification.

This would be enough to satisfy the goals of most developers. However, the lean design of the relationship between the virtual image and the host windowing system provides even more options. As mentioned above, the main point of contact between the virtual image and the host's display mechanisms is the class **GraphicsContext**. In the last issue, we saw that on the input side, the primary point of contact is the **InputState**. Yet, suppose we wanted to display information on a device not supported by R4 or by the host windowing system. All that is required is to provide a subclass of **GraphicsContext** that knows how to draw on that device. The rest of the code in the image will then be able to use that device. If input must be drawn from an alternate source, such as a serial port, then a version of **InputState** could be created that maps the serial input into the input event structures used by Smalltalk. Beyond that translation, no other classes need know that the input is not coming from the usual source.

In a recent experiment, subclasses of **InputState** and **GraphicsContext** were created to allow the Smalltalk user interface to be accessed through standard ASCII terminals. Once the differences in input and output models were hidden in these subclasses, the rest of the Smalltalk environment proved quite robust; very little other code was necessary to get windows up and running on an ASCII terminal. That most of the code in the interface framework works well in such a radically different environment, with most of the effort involved in changing only two classes, is a testament to the elegant design of Release 4. ■

Greg Hendley is a member of the technical staff at Knowledge Systems Corp. His OOP experience is in Smalltalk/V (DOS), Smalltalk-80 2.5, Objectworks\Smalltalk Release 4, and Smalltalk/V PM.

Eric Smith is a member of the technical staff at Knowledge Systems Corp. His specialty is custom graphical user interfaces using Smalltalk (various dialects) and C.

They may be contacted at Knowledge Systems Corp., 114 MacKenan Dr., Cary, NC 27511 or by phone at (919) 481-4000.

BOOK REVIEW

Reviewed by Martin Osborne and Ann Cotton

CONCEPTS OF OBJECT-ORIENTED PROGRAMMING

by David N. Smith
McGraw-Hill, New York, 1991

Have you been hearing a lot of hype about object-oriented programming (OOP) and now you want to see what all the fuss is about? Or are you fairly new to object-oriented programming and wondering if you have the concepts straight? Or are you an object-oriented programming expert tired of trying to explain it to your friends over a cup of coffee? If you answered yes to any of these questions, then *Concepts of Object-Oriented Programming*, by David N. Smith, may be the book for you. Here, the basic ideas and terminology of object-oriented programming are explained and illustrated in a clear, concise, and imaginative manner to the reader who is familiar with the basic concepts of procedural programming. There are a number of concepts associated with object-oriented programming that the book does not discuss. Among these are graphical user interfaces, sophisticated programming environments, object-oriented analysis, and object-oriented design. The book also does not discuss advanced OOP concepts such as delegation, double dispatching, and multiple inheritance.

Mr. Smith is highly qualified to write about OOP. He is a senior programmer and researcher at IBM's Thomas J. Watson Research Center and has been active in OOP since 1983. He is a founding member of the organization that sponsors OOPSLA, the major yearly conference on OOP. Mr. Smith credits the book's origins to a challenge from a friend—give a one foil presentation of OOP. He has never managed to do it, but he has given day-long tutorials on OOP at several major ACM conferences, with this book as the eventual outcome. The slow evolution has allowed Mr. Smith to perfect his presentation of the subject.

In the paragraphs that follow, we will try to give an overview of the book. Brevity will sometimes lead to oversimplification. We will give our opinions at the end.

Mr. Smith writes, "There are many books on object-oriented programming for the professional programmer or designer who wants in-depth knowledge...[but]...there are no books on object-oriented programming for those that simply want to know what it is all about; that just cover the important ideas without trying to make the reader into a programmer or designer of object-oriented systems."

Object-oriented programming has to be understood as a new way of thinking about programming, and its methods are radically different from those of procedural programming. Procedural programming began at a time when small programs resided in core-processed data on cards, and the resulting distinction between data and code remains a part of today's most popular programming languages. Be warned, object-oriented has become a fad term that vendors use freely "to stir some spice into otherwise ordinary products."

Object-oriented programming systems have four interrelated characteristics: encapsulation, inheritance, polymorphism, and typeless variables. In simpler language: data hiding, a hierarchy of object definitions, multiple routines with the same name, and any variable can hold anything. (If your favorite object-oriented programming language does not have all of these features, you might enjoy arguing about the definition.) Throughout the book, the language Smalltalk is used to illustrate the concepts of object-oriented programming. Even people who eventually intend to use a hybrid language such as C++ should start with Smalltalk. However, the intention of the book is to introduce the concepts of OOP, not to train programmers.

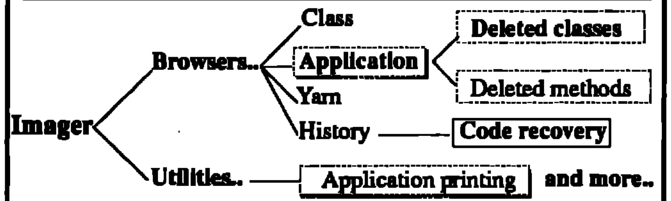
Chairs are familiar objects, and defining one in Smalltalk introduces the notions of instance variables and methods. Instance variables are data, and methods are code; together they combine to form objects. However, code in Smalltalk differs greatly from code in languages such as C and Pascal. A simple sort procedure written in Pascal is only good for sorting one kind of thing, say an array of integers. Sorting an array of real numbers would require another procedure. In Smalltalk, a method implementing the same algorithm is much more powerful. It can sort any collection of things, provided that the things can be compared one to another.

Objects are organized in a hierarchy of classes. Because of this hierarchy, they can inherit instance variables and methods from each other. Inheritance can be used in simple ways or in ways that are subtle and complex (consider inheritance with super and self), all of which are illustrated with concise examples and diagrams. The concept of inheritance is extended "to include abstract classes which are never intended to produce real objects but are used to provide characteristics to be inherited."

The design and implementation of three applications illustrates the real point of OOP. Although small and simple, the applications incorporate a number of the techniques used in designing a good hierarchy of classes and will be of interest to experienced OOP programmers as well as beginners. Design and implementation are seen as being iterative and consist of

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 $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.

defining the hierarchy, protocol, and state of the classes. It is fine to "do what you know how to do...[and]...see if what you just did suggests additional things to do."

Readers who might have been wondering why message and class were not listed as fundamental OOP concepts are told that "the idea of message sending...is a result of other ideas... [and that]...classes are just one way to implement hierarchies." The benefits of object-oriented programming are code reuse, localization of change, design assistance, extensibility, and faster development. For readers who want to pursue OOP further, there is a guide to a few good books, publications, conferences, and software packages.

A reviewer's opinion of a book is bound to depend on her/his background and interests. With this in mind, you are about to receive two sets of opinions. This book is Ann's first exposure to OOP. She has worked as a programmer/analyst on university information systems since 1981 and is currently a database administrator. Martin has been involved with OOP for two years. He teaches computer science, including courses on OOP.

In Ann's opinion, the book is well organized, but the writing is awkward at times. All the concepts are understandable, although sometimes not without repeated reading. Early sections of the book are easy to read, and later ones, more challenging. Technical savvy and a willingness to wrestle with the material are required. She feels that *Concepts of Object-Oriented Programming* has provided her with a good understanding of OOP. However, the subtlety and complexity of some of the examples raise concern that considerable experience with OOP is required before one can think in an object-oriented way and create good object-oriented designs and manageable code. This leads her to question the claims of increased productivity. Skepticism aside, Ann believes that OOP is promising and is wondering how she can take advantage of it in her environment.

Martin wishes that this had been the first book he had read about OOP. If it were, he would have avoided a lot of his initial confusion about OOP. He teaches a class on Smalltalk to students who have had a year of programming experience and he thinks the book provides an excellent checklist of the concepts the class should cover with good examples of how to cover them. He feels that the book overemphasizes the subtleties and complexities of OOP and thus misleads a newcomer into overestimating its difficulties. While all the concepts illustrated are important, they usually do not occur in such density in typical systems. The benefits of OOP stated by the author are ones that most people would agree with and perhaps would add some of their own to. Martin enjoyed Appendix B, which discussed the problems of procedural programming languages. While some might argue that not all the problems related would apply to Ada or Modula, or even other languages when used with thought and care, the list of problems is enough to make one wonder why we continue to program in these languages. But perhaps the current state of affairs is only temporary, and soon most of the major languages will be object oriented or, at least, object enhanced.

We wonder if Mr. Smith found it difficult to decide what to include and exclude, and some readers are bound to wish for a little more of this or a little less of that. However, all in all, the book is excellent and receives two enthusiastic thumbs up. ■

Ann Cotton, after reading Mr. Smith's book, has been introduced to object-oriented programming. Currently, she is the Database Administrator at Western Washington University, Bellingham, Washington. Her interests include relational database technology and computer services. She can be reached at (206) 676-3826 or uis.uucp\!ann@henson.cc.wwu.edu.

Martin Osborne has been involved with object-oriented programming and Smalltalk since 1989. He works in the Department of Computer Science at Western Washington University, Bellingham, Washington, where he teaches classes on object-oriented programming, information systems, and software engineering. His interests include object-oriented programming, visual programming, and software development aids. He can be reached at (206) 676-3798 or martin@cs.wwu.edu

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.

Dome Software Corp. has announced the formation of the Smalltalk Metrics Project. The project's goal is to establish a body of quantitative information about design and development in Smalltalk and to create a better understanding of how it is used in production applications.

Dome has defined a variety of ways to measure programmer activity, program complexity, and component reuse. It has created a Smalltalk program that can compute these measures for a specific system. The program also collects information from the system's developer, including the size and background of the development team, the level of experience of its members, and the methodology used to develop the system.

The company is enlisting the cooperation of other firms by asking them to use Dome's metric program to analyze their own Smalltalk-based systems and report the results to Dome. By gathering data for a wide variety of Smalltalk systems, the company hopes to distill a much clearer picture of how Smalltalk is being used. Companies using Smalltalk to develop production systems are invited to request an information packet from Mr. Wilmes at Dome.

For more information, contact Dome Software Corp., 655 W. Carmel Dr., Carmel, IN 46032; (317) 573-8100

ArchiText, the language-driven document constructor designed to write and maintain structured documents, is now also available for Smalltalk and C/C++.

Developed by Interactive Software Engineering, ArchiText is one of the first products ever constructed using purely object-oriented programming techniques and is written entirely in the Eiffel object-oriented language.

ArchiText is a powerful high-level tool for manipulating and viewing structured documents, such as programs, specifications, and technical or administrative reports. It was designed to relieve computer users from the need to worry about language structures.

A unique feature of ArchiText is its ease of adaptation to any context-free language. To build an ArchiText editor for a specific language (programming language, design language, or even the description of the structure of standardized technical documents), it suffices to describe the language's grammar in a simple notation called Language Description Language (LDL). The standard delivery of ArchiText includes basic LDL grammars for Eiffel, Ada, Pascal, C/C++, and Smalltalk.

ArchiText features a graphical user interface based on the Motif GUI standard that makes it quick and easy to manipulate complex structures.

For more information, contact Burghardt Tenderich, Interactive Software Engineering, Inc., 270 Storke Rd., Ste. 7, Goleta, CA 93117; (805) 685-1006; fax (805) 685-6869.

ParcPlace Systems announced that it will support Information Builders' (IBI) Enterprise Data Access/SQL product family. EDA/SQL provides direct access to information in corporate databases including IBM's DB2 and IMS, Sybase, Oracle, Informix, and IBI. ParcPlace Systems intends to extend its suite of Objectworks\Smalltalk Portable Objects (object-oriented class library) to provide a common interface for applications that use IBI's EDA/SQL product.

ParcPlace Systems also announced that the FACETS\4GL fourth generation language (4GL) application development tool for Objectworks\Smalltalk now includes an interface builder. FACETS\4GL 2.0 enhances the capabilities of traditional 4GLs with a graphical user interface (GUI) builder and provides a migration path from 4GLs to object-oriented technology. FACETS\4GL is developed by Reusable Solutions and marketed by ParcPlace Systems.

For further information, contact ParcPlace Systems, 1550 Plymouth St., Mountain View, CA 94043; (415) 691-6700.

Tigre Object Systems, Inc. announced a new agreement with ParcPlace Systems. Tigre now bundles the Tigre Programming Environment with ParcPlace's Objectworks\Smalltalk object-oriented language. The Tigre Programming Environment, which uses Objectworks\Smalltalk as its scripting language, lets developers create state-of-the-art graphical user interface (GUI) programs that run, without porting, on Macintosh II, Microsoft Windows 3.0, and all popular UNIX workstations.

For further information, contact Tigre Object Systems, Inc., 3004 Mission St., Santa Cruz, CA 95060; (408) 427-4900.

Synergistic Solutions, Inc. announced additional platform support for Smalltalk\SQL, the portable database interface for Smalltalk. The product works in conjunction with ParcPlace Systems Objectworks\Smalltalk to enable graphical user interface (GUI) applications to access information stored in relational databases. Direct database support is currently available for the Sybase and Oracle databases. DB2, Informix, Ingres, Rdb, and other databases may be accessed through a variety of gateway products.

For further information, contact Synergistic Solutions, Inc., 63 Joyner Dr., Lawrenceville, NJ 08648; (908) 855-7634.

InfoWare Version 1.0 is a database connectivity package that provides an object-oriented interface to relational databases for applications written in ParcPlace Systems' Objectworks\Smalltalk. Applications built with InfoWare are able to access relational data in the form of objects. The applications are written entirely in Smalltalk, instead of the embedded SQL of the host server, providing the developer with a uniform object-oriented interface and portability across a variety of RDBMS servers.

For further information, contact Ensemble Software Systems, Inc., 555 Bryant St., Ste. 347, Palo Alto, CA 94301; (415) 325-2773.

WHAT THEY'RE SAYING ABOUT SMALLTALK

Excerpts from industry publications

...As far as object-oriented technology goes, [Bob] Libutti [programming systems director of market strategy with IBM] said that IBM was considering using Sapiens tools internally in several industry areas. And prior to its discussions with Sapiens it has been talking to DigiTalk. Indeed, SmallTalk is used extensively in IBM—it was used to build the new Cross System Product, CSP, and was also used by Intersolv Inc. to build its product line for OS/2. Libutti said that IBM is making a major investment in object-oriented technology. This is all well and good except that IBM's Repository is based on the Entity-Relationship model and not the object-oriented model. However, Libutti says that IBM is extending the Repository to cope with object types, although when pressed said he didn't know when these extensions will appear...

IBM sets the record straight on misperceptions of systems application architecture, ***Unigram X****, 11/18/91*

...Many people hear the term object-oriented and immediately think of a programming language like Smalltalk, C++, or Eiffel. These languages all provide language-level abstractions to create and manage hierarchies of communicating objects that serve as the implementation of user requirements. Some very powerful programming environments have been developed that support these languages by producing special-purpose editors, browsers, and debuggers. These make it easy to deal with complex collections of code. They're bad because they tend to focus on implementation of language issues at the expense of higher-level design or requirements issues...

Languages like C++ and Eiffel support aggregation and inheritance, and consequently are considered true object-oriented languages. Ada supports aggregation, but not inheritance, leading object-oriented purists to call it an object-based language. Still, it's possible to build object-oriented systems with Ada...

Design applications: building a case for object-oriented technology, Read Fleming and Lou Mazzucchelli, ***Electronic Design****,11/7/91*

...Requirement views, context specification, event-object partitioning, environment modeling, design templates, method hierarchies, and a rapid system development strategy are all part of a comprehensive object-oriented approach to system development. All these elements evolved from structured techniques into a significant number of today's OO methods. These methods in various forms have been used and written about extensively for the last ten years. Other concepts required by object-oriented development are readily available from entity-relationship analysis, Jackson's data-structure oriented approaches, and the Smalltalk language. These sources for object-oriented methods have been around for a long time as well. As with elements of the structured techniques, method developers have been incorporating those ideas into what are now known as OO methods. This is not to say that every last issue in object-oriented development methods has been resolved, but the evolution to revolution question is rendered moot given that the structured techniques have already spawned OO methods. Instead of revolution, the real issue is revelation. For a variety of reasons, many in both the object and the structured world have slept through the evolution of OO methods. For all of them, I hope this article is a wake-up call.

Debate: evolution vs revolution: should structured methods be objectified?, John Palmer, ***Object Magazine****, 11-12/91*

...In pure OOP systems, such as Smalltalk, memory allocation and deallocation [are] handled by the language. In C++, as in C, the programmer is responsible for memory management. This complex task is a common source of bugs in C programs.

Techview: strengths and weaknesses of C++, Larry Seltzer, ***PC Week****, 12/2/91*

...When I learned about object-oriented programming using Smalltalk, I was preoccupied by the picayune details of the language. I spent the first six months understanding the subtleties of the syntax, learning the class libraries, studying the language semantics and implementations, and mastering the programming environment. At the end of that first half year, I had a solid grasp on all the little issues of an object language, but I still knew nothing about objects. I had been reimmersing myself in issues familiar to me from my days as a procedural programmer. I focused on the non-object-oriented aspects of my object-oriented language to avoid the uncomfortable feeling that I didn't know what was going on. By clinging to my old ways of thinking, like a nervous swimmer to the side of the pool, I was preventing myself from reinventing my perspective. It was only through patient and expert tutelage that I was able to break free of my old habits and begin to make use of the power in objects. I now know that learning objects needn't be frightening or confusing...If you're willing to trust yourself to learn the syntax and programming environment later (after all, you've probably learned several of each already), you can be doing objects in a few hours, too...

Think like an object, Kent Beck, ***UNIX Review****, 10/91*

...Traditional or "classical" object-oriented languages have chosen a funny spot in the spectrum of possible object-oriented languages. On the one hand, they have taken the model of autonomous objects too far—we could justify calling languages like Smalltalk object-obsessed languages. On the other hand, the designers of object-oriented languages typically have not taken the object-oriented model seriously, and so CLOS is one of the few languages defined as an object-oriented program...

Metaobject protocol: generic functions and methods, Nick Bourbaki, ***AI Expert****, 10/91*

...The Eiffel language is small compared to others, allowing a user to understand and use the entire language instead of a subset. At first glance, it looks like a hybrid between Smalltalk and C. That is not to say the language is a combination of these, but simply that the code will look like a conglomeration of those language's styles and commands. Programming in Eiffel requires an excellent grasp of classes. The terminology used in Eiffel is a collection of terms from other object-oriented languages, which may lead to some confusion. Classes are defined rigorously using the Eiffel language...Learning Eiffel is a little unnerving at first. Experience with other OOP languages will certainly help, but a fair amount of learning is still involved...The difficulty is not the language itself, but the correct use of it. Eiffel is well-defined and, once the concepts are understood, is quite logical. However, a new user can anticipate many days of work before feeling comfortable with it. Once mastered, though, it is easy to remain abreast of the language due to its small size and logical layout...

*Off the shelf: OOP languages, Tim Parker, **UNIX Review**, 10/91*

...Embedded systems development using Smalltalk is no longer a research curiosity. Real systems are being shipped today which have used the technology described...As microprocessors continue to improve and memory becomes even cheaper, the complexity of embedded applications will undoubtedly increase. As this happens, the ability to meet customers' expectations and management's deadlines with traditional tools and methods will decline. Using Smalltalk to develop embedded systems is not a panacea. Developing complex systems will never be easy. But developers who use object-oriented programming systems such as Smalltalk will be engineering high-quality solutions faster and cheaper than their competitors.

*Smalltalk and embedded systems, John Duimovich and Mike Milinkovich, **Dr. Dobb's Journal**, 10/91*

...[BT North America's Mike] Roberts says: "C++ combines the expressive power of object-oriented languages like Smalltalk with the efficiency and low-level control of C. Its wide dynamic range lets you think of objects at abstract levels, then shove bits around when you need to. That's a very attractive combination...Smalltalk was tempting...I like its support of rapid prototyping and hypertext." But the hypertext editor will stand or fall on the basis of its user interface, and Roberts discovered he couldn't live with Smalltalk's. "The object hierarchy that comes with Smalltalk includes a complete graphical user interface[,]... which is very convenient if that's the interface you want. But I'm fussy and Smalltalk's fussy, and our fussiness didn't overlap. I found that to make the changes I wanted to make, I had to dive into the object hierarchy and rewrite code at a very low level. If you're doing that, you might as well be using C++. You aren't enjoying the benefits of Smalltalk's reusable, extensible classes anymore"...C++ is an important tool, Roberts says, but it isn't a panacea. "It is naive to expect that C++ subclass resolution will be useful for all message-routing [in a message-based system]...It is not sufficient for generalized dispatching." Roberts has had to invest significant effort in writing message decoding and dispatching functions for his project...

*Pushing the envelope, J.D. Hildebrand, **UNIX Review**, 10/91*

...At the higher layers, the object orientation of the system is paramount. Designing a PenPoint application is similar to creating one for the Smalltalk environment, in that you instantiate

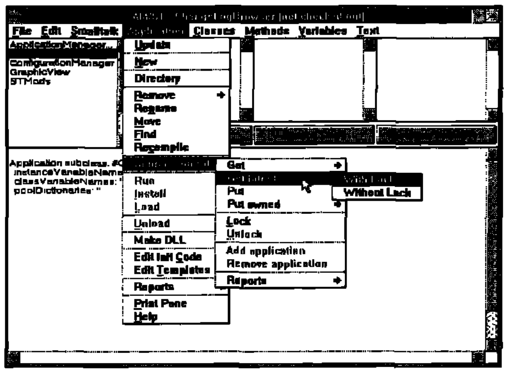
ImageSoft

AM/ST™

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.

C&L | Coopers & Lybrand



Application subclass: #C
instanceVariableNames: ''
classVariableNames: ''
poolDictionaries: ''

Key Features of AM/ST Include:

- Execution tuning & 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.

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

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 and/or subclass the components of your app from existing classes in the application framework...

*A technical overview of PenPoint, R.V., **Dr. Dobb's Journal**, 11/91*

...[Robert Carr, cofounder of GO]:" What we mean when we say that PenPoint is object-oriented is that the programming interfaces are implemented as a sequence of objects that you can send messages to, and that these objects are instances of classes, and classes can be subclassed, thereby modifying their behavior, and of course the objects tend to encapsulate information and behavior and hide the actual data representation from the folks who want to communicate with them. So I think anybody who has studied what I'd call "true object orientation" in the Smalltalk sense will find that when we talk about object orientations that's what we mean...

*A conversation with Robert Carr, part I, Michael Swaine, **Dr. Dobb's Journal**, 11/91*

...How should analysis and design be used with object technology? The answer is: incrementally, in conjunction with actual development, using separate techniques for each layer of a layered system architecture....Instead of designing an entire monolithic system on paper before beginning implementation, sketch out a skeleton system to get you started, then code that much and see how it works. What you learn from the implementation is bound to improve the design. Then take that improved design, sketch out an incremental set of functionality, and roll that into the code. The malleability of object technology makes this incremental approach to development far more feasible that it was with conventional technology. In addition, tons of paperwork can be reduced to mere pounds, the product rolls out faster, and the overall design is usually superior because it is proven in the field and enhanced in stages rather that being thrust upon end users in the conventional "big bang" fashion...

*Easing into objects: redefining analysis & design, David A. Taylor, **Object Magazine**, 11-12/91*

..."[object-oriented programming is] the difference between building a castle out of sand and building a castle with Legos," says Roger Heinen, Apple's vice president of Macintosh Software Architecture...

*Software industry is "object-oriented," Rory J. O'Connor, **Huntsville Times**, 11/10/91*

...Developers using object CASE tools without the prerequisite training, practice, and a fundamental understanding of object will face the same failure as the inexperienced doctor operating on patients using state of the art equipment. Tools alone cannot save the patient!

*Point/Counterpoint: using CASE tools without the methodology, the developer's perspective, Kathleen Meyer, **Object Magazine**, 11-12/91*

[illegible]
That Stores,
Retrieves,
Moves,
Distributes,
Shares,
Protects,
Manages...
OBJECTS

VERSANT
Object Technology

VERSANT is the preeminent object DBMS that gives you:

- High-Performance Object Management
- Efficient Object Level Locking
- Object Level Event Notification
- Object Containers
- Automated Object Migration
- User Defined Object Versioning
- Object Check-out and Check-in
- Runtime Changes to Object Classes
- Bi-Directional Object Linking
- Interfaces for both C++ Objects and Smalltalk Objects

Using an object DBMS with today's object-oriented applications can deliver to you *dramatic* performance improvements, *significant* reductions in code, *simplified* application design and *unprecedented* power and flexibility for managing your application objects.

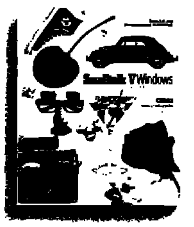
But for you to get these benefits, you need an ODBMS architected to manage objects throughout.

And if your application needs to perform in the demanding environment of distributed, heterogeneous computing – VERSANT is also the natural choice!

To find out why hundreds of users at leading corporations worldwide have chosen VERSANT or to enroll in our next seminar in your area, call 1-800-9OBJECT.

VERSANT
Object Technology

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

Smalltalk/V is a registered trademark of Digitalk, Inc. Other product names are trademarks or registered trademarks of their respective holders.
Digitalk, Inc., 9841 Airport Blvd., Los Angeles, CA 90045
(800) 922-8255; (213) 645-1082; Fax (213) 645-1306

LOOK WHO'S TALKING

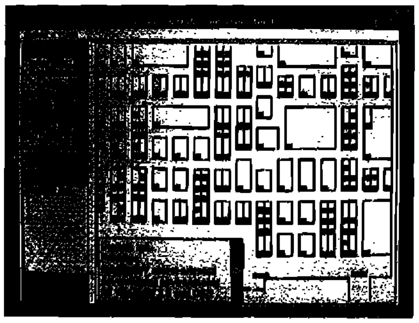
HEWLETT-PACKARD
HP has developed a network troubleshooting tool called the Network Advisor. The Network Advisor offers a comprehensive set of tools including an expert system, statistics, and protocol decodes to speed problem isolation. The NA user interface is built on a windowing system which allows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