The Smalltalk Report

The International Newsletter for Smalltalk Programmers

September 1992 Volume 2 Number 1

Experiences with Smalltalk on a Large Development Project

By Bran Selic

Contents:

Features/Articles:

1 Experiences with Smalltalk on a Large Development Project *by Bran Selic*

8 SmallDraw—Release 4 Graphics and MVC, Part 3 *by Dan Benson*

Columns:

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

17 *Getting Real:* Extending the Collection Hierarchy *by Juanita Ewing*

19 *Smalltalk Idioms:* ValueModel Idioms *by Kent Beck*

Departments:

23 Product News & Highlights

One of the most frequently asked questions about object-oriented technology is whether it was used as the primary technology on a large project. This question is particularly relevant to Smalltalk because it is often said that Smalltalk is a language well-suited for prototyping but not for "real" product development. In this article we will describe our experience using Objectworks\Smalltalk from ParcPlace Systems as the basic implementation language for a commercially available CASE tool called ObjecTime. This project is currently in its sixth year and at one point involved over 30 Smalltalk programmers.

THE PRODUCT

Bell-Northern Research (BNR) designs and develops real-time distributed telecommunications systems for its parent company, Northern Telecom. The software driving these systems is often surprisingly complex and usually involves many millions of lines of high-level code. To meet the extreme quality and robustness requirements of such systems, it is obvious that powerful computer-based development tools are required. ObjecTime (previously known as Telos) is one such CASE tool created at BNR for constructing the next generation of distributed event-driven systems. It can be used for analysis, design, implementation, and verification. The tool is a key component of a methodology called Real-Time Object-Oriented Modeling (ROOM), which is characterized by a set of high-level design paradigms and a highly iterative development process.[1] With ObjecTime, users graphically capture the high-level aspects of their designs and combine them with specifications written in C++, or a simple rapid prototyping language for the more detailed aspects. These designs can be executed directly using ObjecTime's built-in run-time environment. ObjecTime is currently the most widespread CASE tool within BNR. It has been made available to external (non-BNR) customers and has already been purchased by several major corporations.

The software comprising the tool is quite elaborate and includes an interactive graphical user interface, several complex semantic editors, a high-level language compiler, and an event-driven run-time system. This system's level of complexity can be deduced from the size of the class hierarchy, which currently contains close to 1,400 Smalltalk classes.

THE PROJECT AND ITS CHRONOLOGY

The project has so far progressed through three principal stages: a prototyping stage, a development stage, and a commercial product stage.

The prototyping stage

The prototyping stage started in late 1986 and lasted approximately 18 months, during which time the project team grew from three to 18 people. None of the

continued on page 4...

John Pugh

Paul White

EDITORS' CORNER

HAPPY ANNIVERSARY! We thought somebody should say it, as we roll into year two of THE SMALLTALK REPORT. We trust you have been satisfied with the quality of articles over the past 12 months. Subscriptions are constantly climbing, as is the number and diversity of Smalltalk users. We have tried to include articles that have a broad band of appeal yet are specific enough to give you more than just a "warm feeling." Certainly the best part of this job has been the opportunity to meet many of you (albeit electronically in most cases!!). Please, keep coming forward with ideas.

As you are all aware, one requirement sorely lacking in our niche of the software industry is a repository of documented experience reports. Other than OOPSLA's experience reports, very little is available in terms of actual documented case studies. Newcomers to object-oriented technology, and Smalltalk in particular, want to see proof that the technology has been successful. And those of you trying to get on with the development of software know how much easier life would be with a reservoir of experiences from previous projects, both good and bad, on which to draw. If you're like us, you're constantly left with the feeling that "this has been done before," especially in terms of adapting traditional management strategies to Smalltalk projects. It's time we started to reuse more than just code.

Bran Selic's feature article describes many experiences gained during the development of the CASE tool ObjecTime at Bell Northern Research. He gives a chronology of the project, highlighting things that worked well and some of the ptifalls encountered.

Also in this issue, Dan Benson concludes his three-part series on the development of SmallDraw, his graphics editor, illustrating the "ins and outs" of MVC. He adds facilities to SmallDraw to allow grouping, layering, and alignment of objects, cut/copy/paste facilities, and scrolling.

Three of our regular columns appear this month with each building on themes developed in earlier columns. Kent Beck's column describes the inherent shortcomings of the change propagation mechanism and describes the ValueModel style of coding introduced in Objectworks\Smalltalk 4.0. Juanita Ewing continues her discussion of proper use of inheritance through an example of adding an OrderedSet to the Collection hierarchy. Finally, Alan Knight continues his survey of many of the complaints registered on USENET about OOP.

In closing, we would like to take the opportunity to thank those of you who have helped us out over the past year. A special thanks goes to our regular columnists, who have yet to let us down and whose contributions form the pillar of the REPORT. Thanks, gang!

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

Submit articles to the Editors at Smalltalk Report, 91 Second Avenue, Ottawa, Ontario K1S 2H4, Canada.

The Smalltalk Report

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

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

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

Columnists
Kent Beck, First Class Software
Juanita Ewing, Digitalk
Greg Hendley, Knowledge Systems Corp.
Ed Klimas, Linea Engineering Inc.
Alan Knight, Carleton University
Suzanne Skublics, Object Technology International
Eric Smith, Knowledge Systems Corp.
Rebecca Wirfs-Brock, Digitalk

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

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

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

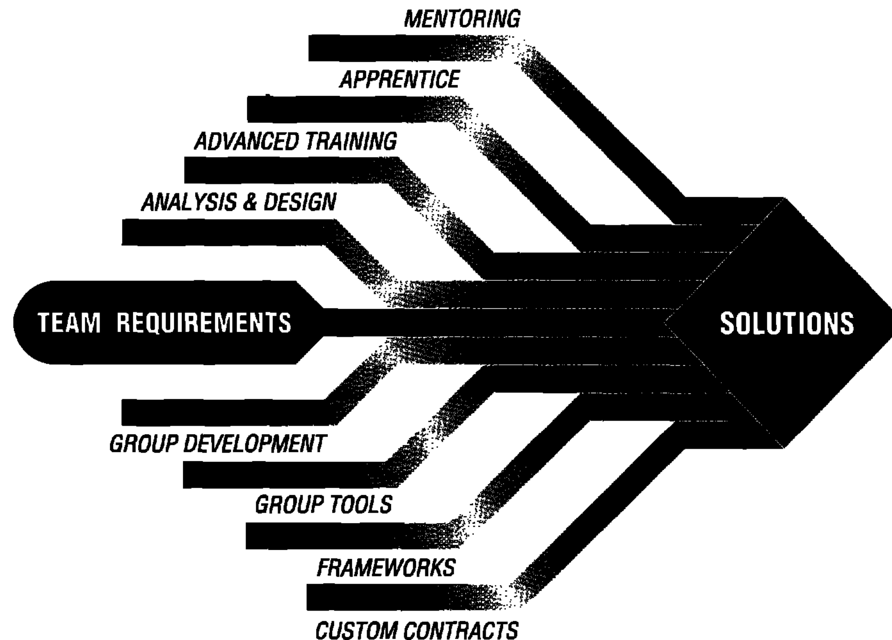
Marketing/Advertising
Diane Morancie, Advertising Mgr—East Coast/Canada
Holly Meintzer, Advertising Mgr—West Coast/Europe
Geraldine Schafran, Exhibit/Recruitment Sales Manager
Sarah Hamilton, Promotions Manager—Publications
Lorna Lyle, Promotions Manager—Conferences
Caren Polner, Promotions Graphic Artist

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

Margherita R. Monck
General Manager

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

Transition to Object Technology by Design



The Management Challenge

The transition to object technology must be designed for success. The management challenge is to:

- Produce Quality Software
- Deliver on Time
- Build Maintainable Code
- Model the Business Problem
- Build Client-Server Solutions
- Manage Complexity

Knowledge Systems Meets the Challenge

Knowledge Systems Corporation (KSC) has emerged as the industry leader in delivering pure object-oriented product solutions. KSC products and services are designed to successfully transition business to object technology.

Transition Services

KSC Transition Services include contract services and a complete training curriculum that supports a group development environment. Multiple training tracks are designed to ultimately attain self-sufficiency and to produce deliverable solutions. Program curriculum includes:

- Mentoring: Process Support
- Apprentice: Small Group Project Focus at KSC
- Finding the Objects (CRC)
- OO Analysis and Design
- Introductory to Advanced Programming in Smalltalk
- Introduction to Smalltalk for COBOL Programmers

Development Environment

KSC now markets in the U.S. and fully supports ENVY™/Developer, a multi-user development environment. In addition, KSC provides integrated services and tools to enable construction of cooperative processing applications.

Design your Transition

Begin *your* successful transition to object technology today. Join the growing list of KSC clients such as IBM, Hewlett-Packard, Texaco, Fisher Controls, American Airlines, First Union, Northern Telecom, and Texas Instruments. For more information on transition products and services from Knowledge Systems, call us at 919-481-4000.

Knowledge Systems Corporation

OBJECT TRANSITION BY DESIGN

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

© 1992 Knowledge Systems Corporation. ENVY is a trademark of Object Technology International, Inc.

...continued from page 1

team members had practical experience with O-O technology but we decided to adopt an O-O approach.

Communications software traditionally has been designed using an object-based approach, primarily because of the inherently distributed and asynchronous nature of communications systems. We were looking for a new technology that could overcome some of the major limitations of traditional software construction methods.

After some deliberation, we chose Smalltalk as the implementation language for our prototyping. Various object-oriented flavors of C (Objective C, C++) were also considered and discarded. We felt that a qualitatively different technology was required to deal with the complexity we had forecast for the coming generation of software systems. We were interested in programming abstractions that could deal with entire subsystem architectures and complex graphics. The semantic gap between these and the low-level machine-oriented abstractions provided in C and similar languages was just too great.

We originally selected Smalltalk/V from Digitalk Inc. After about a year, we switched to Smalltalk-80 from ParcPlace Systems because ParcPlace software ran on the Unix-based workstations used by most of our client base. In addition, our own performance benchmarks indicated that at that time (late 1987), our application would execute more than twice as fast on ParcPlace Smalltalk than on Smalltalk/V on the same platform. The port of our code to Smalltalk-80 was straightforward with most of the difficulties stemming from differences in the graphics paradigms.

There was no formal design process but the issue was discussed at length, with great fervor and some dissent. The highly interactive Smalltalk development environment was unlike any the team had experienced before. It obviously had great potential that was not exploited fully by traditional linear models of software development.

Our initial development consisted of a set of disjoint prototypes of different toolset components, each one designed and implemented by a single developer. In the latter part of the prototyping stage the distinct components were integrated, one-by-one, into a composite whose functionality roughly approximated that of the desired system. There were no commercially available team programming environments at that time so we eventually evolved a "manual" process for synchronizing the activities of programming teams.

This process was based on a weekly integration cycle. At the beginning of each week a new version of the system was generated by the system integrator. Once this image was available, designers would copy it to their own environment and make further changes to it as necessary. At the end of the week, designers would submit their changes for inclusion in next week's image. To minimize conflicts, all the classes in the hierarchy were partitioned so that each class was owned by a group. Only members of the group owning a class were allowed to submit changes for that class. Also, it was possible to specify the integration order of a submission relative to other submissions. A common "patches" repository was maintained for any changes that needed to be shared in the interval between successive integrations. These could be filled in at the discretion of the individual developer.

To our surprise, we found that this manual process was effective even in later stages of the project when the development team was much larger. We attributed this to the decoupling effect of partitioning the class hierarchy across different groups as well as to the highly modular and loosely coupled architecture of the application.

The development stage

Following our prototyping experience we commenced the actual implementation in September of 1988. This second stage lasted approximately two years. During that time the internal architecture of the tool was reorganized and almost all of the prototype code rewritten. The development team doubled in size to eventually include over 30 developers (not including managers), all of them programming in Smalltalk.

The software was developed gradually, in four successive releases, each release extending the capabilities of the previous one. One of those releases included porting of the complete software from a Macintosh platform to a Unix workstation (Sun Microsystems SPARCstation 1). This porting effort turned out to be trivial despite significant differences between the underlying hardware and operating systems. The ease with which this was accomplished confirmed the portability claim of the ParcPlace Systems Objectworks\Smalltalk product.

A more formal development process was used during this stage since we were working on a production version of the software and a much larger team was involved. The final version of this process is described in a later section.

The commercial product stage

Until the end of 1990 ObjecTime was exclusively targeted to internal BNR projects. In 1991 the potential for more widespread use was recognized and a decision was made to market the technology. This meant setting up a full-fledged support organization, "robustification" of the software to commercial-quality standards, creation of high-quality user documentation, and functional extension with features required by a much wider open market. With basic toolset architecture and functionality in place this was accomplished by a smaller and more focused team.

The current release of the toolset, ObjecTime Release 4.0, contains close to 1,400 classes and the initial image requires 5.8 MB. Despite these relatively large numbers, we have not yet encountered nor do we anticipate any fundamental technical or resource limitations of either the language or the ParcPlace Objectworks\Smalltalk environment.

EXPERIENCE WITH SMALLTALK

This section summarizes some of the salient aspects of our Smalltalk experience.

continued on page 6...

VISIT OUR BOOTH AT OOPSLA!

10 Years Ago, When OTI Suggested That Object-Oriented Technology Would Revolutionize The Software Industry, People Called Us Crazy...

Now, They Simply Call Us.

For over 10 years, OTI has been on the leading edge of object-oriented software engineering. And today, as more and more companies adopt this exciting, new technology, OTI remains the leader in providing industrial and commercial object-oriented solutions.

Partners in Object-Oriented Development

OTI's unique technology alliance program provides a means of accelerating product development and introducing new software technology. OTI's technology is being used in products ranging from pen computers to real-time systems. Through these alliances, we've earned a solid reputation for developing high-quality, reliable software – on-time, within budget and to demanding product specifications. This success is attributed to OTI's ENVY®/*Developer* – the first multi-user development environment for object-oriented engineering.

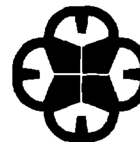
OTI's ENVY/*Developer* – Product Development Tools For Smalltalk

With ENVY/*Developer*, large and small software engineering teams work within an interactive, shared programming environment. Inside this environment, team members share common development tools, common software components and common source code – that means faster cycle times, increased productivity, virtually no duplicated code, and no wasted effort.

Applications are created efficiently and effectively, from beginning to end. Using ENVY/*Developer*, the team passes the application through each phase of the software manufacturing lifecycle – conceptualizing, prototyping, manufacturing, testing, release and maintenance – without ever leaving the environment. ENVY/*Developer* also tracks this process by providing complete software version control and multi-platform configuration management.

Interested?

If your organization is interested in joint research and development or you would like more information on ENVY/*Developer* and object-oriented programming environments, call us today.



Object Technology International Inc.
Engineering Ideas Into Products

Canada Telephone: 613-820-1200 • Fax: 613-820-1202 • E-mail: info@oti.on.ca USA Telephone: 602-222-9519 • Fax: 602-222-8503
ENVY is a registered trademark of Object Technology International Inc.

continued from page 4...

Productivity

We are convinced that Smalltalk, with its sophisticated and customizable environment, source-level debugging capability, extensive class library, and automated storage reclamation, is significantly more productive than most other development environments (including, to a lesser degree, other O-O environments).

This is substantiated to a certain extent by an interesting case that occurred during the project. As part of our development we were required to implement a general purpose graphical windowing system using Objectworks\Smalltalk. Simultaneously, a second development group was independently implementing a similar facility in C based on an X Window System toolkit. This substantial application amounted to approximately 66,000 lines of C code, while the same functionality in Smalltalk required only 6,200 lines of Smalltalk—a functionality ratio of 10 to 1 per line of code! A more conservative estimate, based partly on these results and partly on our overall experience on this project, is that Smalltalk gave us a productivity advantage three to five times over a traditional programming language such as C.

We believe that Smalltalk has a significant productivity edge over other O-O languages as well. Although we have no hard quantitative data, our rough estimate is that Smalltalk is at least two to three times more productive than C++.

Performance

ObjecTime is a computing-intensive application: It has a graphical interactive user interface, it must perform complex semantic checks in real time, and it must efficiently execute complex high-level designs. By far the greatest portion of this functionality is implemented in Smalltalk. (Lesser portions [approximately 5%] were implemented in C++, not for performance reasons, but to enable execution of the C++ segments of a user's design.) Although we occasionally encountered performance problems, in most cases we were able to improve performance to acceptable levels either via straightforward code optimization or through readjustment of the architecture.

The only potentially serious problem relating to performance is an occasional pause for memory compaction, which is part of the automatic garbage collection mechanism. For our application, we found that this pause becomes unacceptable in situations where there is not enough real memory so part of the garbage collection involves swapping memory from disk. To eliminate this problem we stipulated a minimum amount of real memory for our application. Memory requirement is a function of the size of the user design. For ObjecTime release 3.5.1, minimal memory requirement starts at 16 MB (on a Unix workstation) for small to intermediate designs and goes up to 40 MB for the largest designs. With sufficient memory in place, the garbage collection pause is relatively short (between 4 and 10 seconds) and occurs infrequently (every 15–20 minutes).

Quality

Most of our development was done with the ParcPlace Systems product, Objectworks\Smalltalk (from release 2.1 through release 2.5). In over four years we encountered only two problems, both minor, which required product fixes by the vendor.

Usability for large system development

Our experience demonstrated that Smalltalk was a practical solution for moderately large development teams (30 programmers) even without the assistance of specialized team programming tools. Of course, if such tools are available (e.g., ENVY/Developer from Object Technology International), they should be used, since they add significant value and can extend the applicability of Smalltalk to even larger projects than ours.

Training

Carleton University is one of the major world centers of Smalltalk expertise. The School of Computer Science at Carleton organized a short course, taught by professors John Pugh, Wilf LaLonde, and Dave Thomas, which for most team members was the initial exposure to Smalltalk. We were also able to hire, on a temporary basis, a group of graduate and undergraduate students who served as consultants on proper Smalltalk usage. The presence of such experienced Smalltalk programmers significantly cut down on our training time.

In addition to the Carleton course, we took an "intermediate" level Smalltalk course offered by ParcPlace Systems, which focused on common techniques for effective usage of the environment. This course visibly increased the confidence level of the development team.

It takes between one and three weeks for an experienced programmer to learn enough Smalltalk to start using it on the job. However, for a programmer to effectively use Smalltalk, it is necessary to become familiar with the O-O paradigm, the class library, and the programming environment itself. In our experience the majority of programmers needs an additional 6 to 20 weeks to reach an "intermediate" level of proficiency. (Keep in mind that the same amount of time is needed to learn the environmental particulars [e.g., code libraries] for any large project.)

The development process

Our development process differed somewhat from the traditional model. First of all, we wanted to take advantage of the rapid prototyping capability of Smalltalk. Proper use of this feature helps designers gain valuable insight early in the development cycle and before major implementation effort is expended. Inheritance also adds a new aspect to the overall design effort. Typically this requires additional effort consisting of another pass through the design after the desired functionality is fully achieved. Further design optimization is accomplished from the perspectives of reuse and abstraction. We ultimately settled on a process consisting of four main activities:

1. *Functional design* defines the functionality of the feature being developed. The output of this activity is a Functional Specification document which can be discussed with clients. Once finalized, this specification is also given to an

independent verification group to allow early preparation of test plans.

2. *Object or class design* is the fundamental synthesis process in which a high-level design is worked out for the feature. If the feature is complex enough, a formal Design Document is produced for review purposes.
3. Coding is part of the *prototyping and refinement* activity. In the case of prototyping, this activity is often concurrent with and supplemental to class design and even functional design. Given the importance of user interfaces to our application, a distinct subactivity is early modeling and evaluation of the user interface design.
4. *Documentation and testing* are usually done in the final stage. Each designer generates a functional test plan that is reviewed and used for white box testing. For major features, code inspections are also held. This phase also includes testing of the software by an independent verification group.

Although the individual activities are listed in sequence, the process allows for internal cycles to accomodate further refinements, particularly following implementation.

The project management process

The iterative nature of the development process makes it difficult to detect whether or not it converges. To get around this we specified a linear progression of milestones, each one tied to a concrete deliverable. The interval between successive milestones was fixed in advance, based on a priori estimates of the effort required. For example, the formal release of a Functional Design document was the first milestone following the start of feature development. Other major milestones included the release of an Object Design document, the delivery of code to a test group, and the successful completion of testing. Not surprisingly, we had the most difficulty estimating the amount of effort needed for individual milestones to be achieved. This was especially problematic at the beginning because we had had no previous experience with an iterative development process or the O-O paradigm.

Additional observations

To conclude this summary of our experience, we list several additional points pertaining to O-O development:

1. The management team must have an in-depth understanding of O-O technology to gain maximum return from it. This technology is different enough from traditional ones (e.g., the focus on reuse, iterative development process) that many of the long-established management practices are inappropriate. Because this is a relatively new technology not many technical managers are experienced with it.
2. There is a significant need to develop better management metrics to reconcile an iterative development process with the needs of management so that a process stays within its allocated resources. Successive refinement can indeed reach a point of diminishing returns. How do we detect when that point has been reached? New metrics are also required to measure productivity; with refinement, the number of lines of code can actually decrease with time through inheritance and reuse.
3. The ease and rapidity with which code can be changed and recompiled in Smalltalk can easily lead to hacking with little or no time taken to reflect. (Smalltalk is one of those seductive environments where it is very easy for the medium to become the message.) This style of development tends to work bottom up and does not extend very well to large system design. The best way to avoid this is to ensure that a system architecture is defined before any development of details takes place.

anyDeveloper at: AMIX make: money

Just opened!

The first online Smalltalk marketplace where any developer can sell or buy Smalltalk tools, components, add-ons, advice or training, and hook up with the right people. If you're looking for the best in Smalltalk, come to the AMIX online marketplace.

We're offering the AMIX software for free. Visit the AMIX Booth (#701) at OOPSLA, October 18-22 in Vancouver. Or call us now at 415-903-1000 and we'll send you a disk today.

American Information Exchange Corporation
1881 Landings Drive
Mountain View, CA 94043-0848
Phone: 415-903-1000
FAX: 415-903-1093

AMiX

CONCLUSION

We have been using Smalltalk on our project for almost six years; overall, our experience remains strongly positive. We have confirmed not only that Smalltalk is powerful and robust enough to be used for commercial-quality software, but also that there are substantial benefits when compared with other implementation options. Finally, we have demonstrated that Smalltalk can be used successfully on large and long-term projects involving sizable programming teams. ■

References

1. Selic, B., G. J. Gullekson, and I. McGee Engelberg. ROOM: An Object-Oriented Methodology for Developing Real-Time Systems, Montreal, Canada, July 6–10, 1992.

Bran Selic is Senior Manager responsible for real-time CASE technology at Bell-Northern Research in Ottawa, Canada. He can be reached at 613.763.3954 or at selic@bnr.ca.

SMALL DRAW — RELEASE 4 GRAPHICS AND MVC, PART 3

Dan Benson

SmallDraw is a simple structured graphics editor that provides an example of graphics rendering and MVC application construction in Smalltalk-80 Release 4. The first article in this series contained an introduction to graphics concepts and application construction with the MVC architecture through the definition of a "minimal" SmallDraw. The second article added the ability to select and modify objects in the view. This third and final article extends the features of SmallDraw to include grouping of objects, layering of objects, alignment of objects through a DialogView, cut/copy/paste operations through a shared clipboard, the use of command keys, and scrolling of the view. Information on obtaining the complete source code for SmallDraw is given at the end of the article.

GROUPING OBJECTS

Grouping objects together allows them to be treated as a single unit. That is, a grouped collection of objects can be translated, scaled, and copied as a single object. To do this, a new class is defined as a subclass of SDGraphicObject, called SDGraphicGroup:

```
Object ()
    SDGraphicObject ('insideColor' 'borderColor' 'lineWidth' 'handles'
        'boundingBox')
        SDGraphicGroup ('elements')
```

SDGraphicGroup's single attribute, elements, holds a collection of SDGraphicObjects. It implements specific methods for calculating its boundingBox, displaying its elements, testing for point inclusion, and translation and scaling. For example, SD-GraphicGroup defines the following method for translation:

```
translateBy: aPoint
    self elements do: [:o | o translateBy: aPoint].
    self computeBoundingBox
```

The SmallDraw model is responsible for grouping objects. When the group operation is selected from the menu, SmallDraw creates a new SDGraphicGroup, setting its elements to the currently selected set of objects. The selected objects are removed from SmallDraw's objects and the new SDGraphicGroup is added to SmallDraw's set of objects.

The inverse operation of un-grouping is also provided. When this operation is selected, SmallDraw removes any instances of SDGraphicGroup from the current selection, adding each individual element to its set of objects.

LAYERING OBJECTS

As objects are added to the drawing they are placed on top of existing objects; that is, they are conceptually layered. This idea is also reflected exactly in the SmallDraw objects instance variable as an OrderedCollection of objects.

It is often useful to change the relative positioning of objects within the stack. This is accomplished by providing four menu selections, shown in Figure 1, for moving objects to the front or back of the stack, or forward or backward by one position.

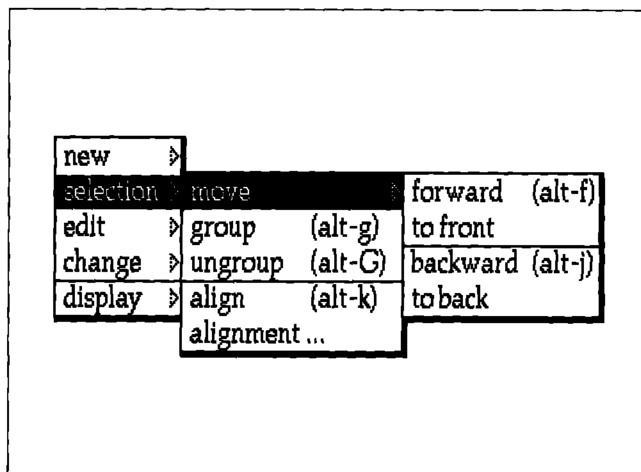


Figure 1. Menu selection for moving objects.

Moving selected objects to the front is done by simply removing them from the list of objects and adding them to the front of the list:

```
moveToFront
    self hasSelection ifTrue: [ | selection |
        selection := self selectedObjectAssociations.
        selection do: [:oa | self objects remove: oa].
        self objects addAllFirst: selection.
        self changed: #rectangle with: self selectedObjectsDisplayBox]
```

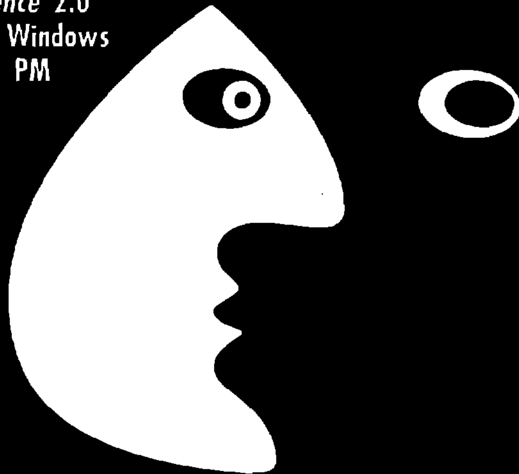
Moving objects forward by one position is done by inserting the selected object before the object that was in front of it:

```
moveForward
    self hasSelection ifTrue: [
        self selectedObjectAssociations do: [:oa | | before |
            self objects first == oa
                ifFalse: [before := self objects before: oa.
                    self objects remove: oa.
                    self objects add: oa before: before]].
        self changed: #rectangle with: self selectedObjectsDisplayBox]
```

Moving objects to the back or backward one position is done in a similar fashion.

s·i·l·e·n·c·e

Now available!
silence 2.0
for Windows
and PM



Multi-user source code control
and versioning system
for Smalltalk/V

NEW! code managed on a client-server model •
NEW! automatic background updating •
NEW! linked sub-project support •
NEW! UFO persistent object toolkit •
NEW! Automatic report generation •
automatic change documenting •
ship compiled code without source •
package and lock releases •
change log browser and restorer •

Starting from
$149.95
source code included

digamma solutions
Unit 6, 387 Spadina Avenue, Toronto, Ontario, Canada, M5T 2G6 Phone: (416) 351-8833 Fax: (416) 408-2850 CompuServe 75430,400
Shipping and handling $15.00 mail, $25.00 courier inside North America. $25.00 mail, call for courier price outside North America. Visa orders add 5%. NO AMEX OR MASTERCARD.
Canadian orders add 7% G.S.T. Ontario orders add 8% P.S.T. **silence** is a trademark of digamma solutions. Smalltalk/V is a registered trademark of Digitalk, inc

ALIGNING OBJECTS

A difficult and time-consuming task in any graphics editor is trying to get objects aligned with each other. Confining the mouse to a low-resolution grid is helpful but not always adequate. This task can be simplified with the use of a DialogView to specify the type of alignment desired. Alignment can take place in either of two directions and one of three positions for each direction (see Figure 2).

The user has the option of choosing one or both directions. For each direction, only one position can be specified using the radio buttons. The chosen alignment positions are retained by SmallDraw so that they may be applied to selected objects without bringing up the DialogView each time. Therefore, two menu selections are added, one for applying the current alignment and one for setting the stored alignment.

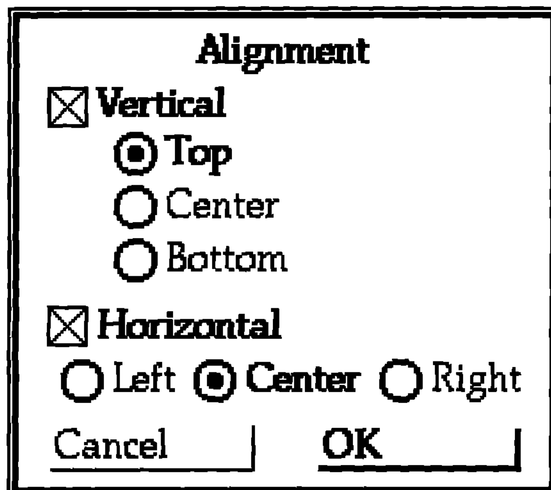


Figure 2. Alignment Dialogview.

When the alignment is to be set, SmallDraw creates a DialogView whose model is SmallDraw. When the DialogView is opened, SmallDraw specifies a message selector (#finishedAlignment) that determines when the view should be closed. Until that message selector returns true, the DialogView interacts with the user and SmallDraw to set and modify the alignment directions and positions.

The vertical and horizontal positions are represented as symbols. These values are stored along with a flag that indicates whether Cancel or OK was pressed in the DialogView. Rather than adding three new instance variables to SmallDraw, a single instance variable called alignment is added. This is an instance of a three element Array to store the three pieces of information as follows:

```
initializeAlignment
    "The alignment instance variable is an array of three elements:
    1) vertical alignment | nil
    2) horizontal alignment | nil
    3) false | true | nil -> cancel | accept | not finished (used by
       DialogView)

    The last flag must be set to nil each time the DialogView is opened.
    See openAlignmentDialog and finishedAlignment."
```

```
alignment isNil
    ifTrue: [alignment := Array with: nil with: nil with: nil].
alignment at: 3 put: nil
```

Methods are used to access the alignment array elements as follows:

```
acceptAlignment
    alignment at: 3 put: true
acceptedAlignment
    ^alignment at: 3
cancelAlignment
    alignment at: 3 put: false
finishedAlignment
    ^(alignment at: 3) notNil
horizontalAlignment
    ^alignment at: 2
horizontalAlignment: aSymbol
    alignment at: 2 put: aSymbol.
    self changed: #horizontalAlignment
verticalAlignment
    ^alignment at: 1
verticalAlignment: aSymbol
    alignment at: 1 put: aSymbol.
    self changed: #verticalAlignment
```

Alignment is performed relative to the total boundingBox of the currently selected set of objects:

```
doAlignment
self hasSelection ifTrue: [| bb repair |
    bb := self selectedObjectsBoundingBox.
    repair := self selectedObjectsDisplayBox.
"Vertical movement."
    self verticalAlignment = #top ifTrue:[
        self selectedObjects do: [:o | o translateBy:
            0@(bb origin y - o boundingBox origin y)]].
    self verticalAlignment = #center ifTrue:[
        self selectedObjects do: [:o | o translateBy:
            0@(bb center y - o boundingBox center y)]].
    self verticalAlignment = #bottom ifTrue:[
        self selectedObjects do: [:o | o translateBy:
            0@(bb corner y - o boundingBox corner y)]].
"Horizontal movement."
    self horizontalAlignment = #left ifTrue:[
        self selectedObjects do: [:o | o translateBy:
            (bb origin x - o boundingBox origin x) @0]].
    self horizontalAlignment = #center ifTrue:[
        self selectedObjects do: [:o | o translateBy:
            (bb center x - o boundingBox center x) @0]].
    self horizontalAlignment = #right ifTrue:[
        self selectedObjects do: [:o | o translateBy:
            (bb corner x - o boundingBox corner x) @0]].
    self changed: #rectangle with: repair]
```

CUT/COPY/PASTE

A common metaphor in many applications is the cutting, copying, and pasting of objects using a "clipboard" as an intermediate storage mechanism. The Macintosh system is an excellent example of using a common system clipboard to transfer a variety of data objects between applications. Similarly, graphic objects can be copied or cut to a common buffer accessed by all SmallDraw applications.

Intermediate storage implies an instance variable that can reference collections of graphic objects. Sharing access to this storage among SmallDraw instances suggests that a SmallDraw class variable is the appropriate mechanism for a common clipboard. Therefore, a class variable called Clipboard is added to the SmallDraw class. The Clipboard can hold one object, or one collection of objects, at a time. Copy and cut operations are destructive because they overwrite the current contents of the Clipboard. Pasting is nondestructive because a copy is made of the Clipboard contents and added to the drawing.

It may seem trivial to implement the copy operation by simply assigning the Clipboard class variable to a copy of the selected objects:

```
copy
    self hasSelection
        ifTrue: [Clipboard := self selectedObjects copy]
```

However, care must be taken when copying and pasting objects to and from the Clipboard. The Smalltalk copy performs a shallow copy, which simply duplicates references to the objects to be copied (making them identical and thus equal), and the Clipboard then points to the objects remaining in the drawing. In contrast, a deepCopy creates exact duplicate objects that are different from the originals (equal but not identical):

```
copy
    self hasSelection
        ifTrue: [Clipboard := self selectedObjects deepCopy]
```

It is not necessary to use deepCopy when objects are cut from the drawing. In this case, the objects are removed from the drawing and essentially transferred to the Clipboard:

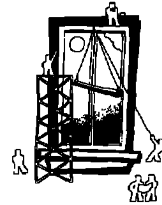
```
cut
    self hasSelection ifTrue: [
        Clipboard := self selectedObjects.
        self objects: (self objects reject: [:p | p value]).
        self changed: #rectangle with: self clipboardDisplayBox]
```

When objects are copied to the Clipboard, they retain their attributes including their location in the drawing. A copied object immediately pasted back into the drawing covers its original copy. A useful convention is to paste an object into the drawing at an offset from its copied position. Each subsequent paste of the same object would then be offset from the previous pasted object. This can be accomplished by defining a paste offset constant and translating the contents of the Clipboard with each paste operation:

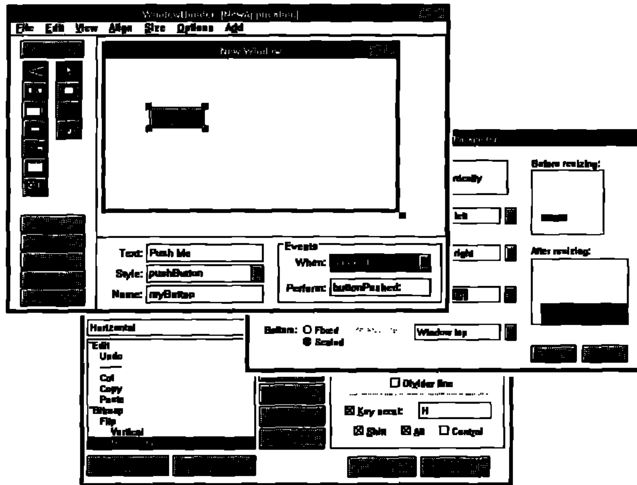
```
pasteOffset
    "Answer the default offset for pasting objects from their copied
        positions."
    ^10@10

paste
    self clipboardFull ifTrue: [
        self deselectAll.
        self objects addAllFirst: ((Clipboard do: [:o |
            o translateBy: self pasteOffset])
                deepCopy collect: [:o | o -> true]).
```

COOPER PETERS

WindowBuilder



The Interface Builder for Smalltalk/V



"... 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.

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

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


```
self changed: #rectangle with: self clipboardDisplayBox]
```

Note that all pasted objects become the current selection by setting the value part of the Association to true. Making duplicates of objects can be simplified by defining a duplicate operation that bypasses the Clipboard:

```
duplicate
"Add a copy of the current selection without changing the Clipboard."
self hasSelection ifTrue: [ | newObjects |
   newObjects := (self selectedObjectAssociations deepCopy do: [:oa |
      oa key translateBy: self pasteOffset]).
   self deselectAll.
   self objects addAllFirst: newObjects.
   self changed: #rectangle with: self selectedObjectsDisplayBox]
```

COMMAND KEYS

As an input device, the mouse is a convenient mechanism when working with modern bit-mapped graphical user interfaces. However, it is often faster and less tiring to perform a command via the keyboard than to make a selection from a menu.

Keyboard commands are distinguished from normal typing by pressing a combination of two keys: the command key and a letter key. The command key looks like ⌘ on the Macintosh and is the alt key on the IBM RS/6000. Other platforms may vary. The Smalltalk class InputSensor refers to the command keys as *alt* or *meta* (depending on the platform) and responds when either is pressed through the messages altDown and metaDown, respectively.

Command key equivalents can be defined for most of the operations that SmallDraw performs. Borrowing from a popular commercial structured graphics application, the following keys are used to invoke the following operations:

key	operation
x	cut
c	copy
v	paste
f	move forward
j	move backward
d	duplicate
a	select all
k	align
g	group
G	un-group

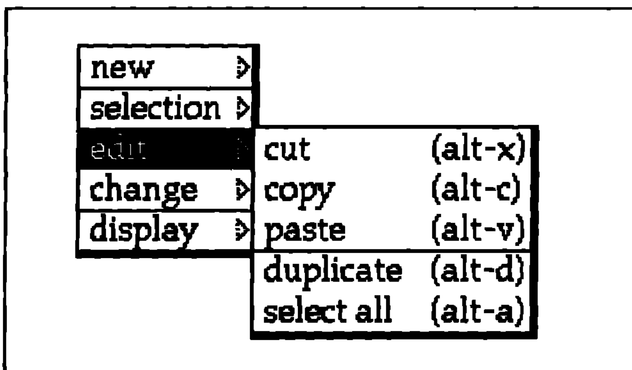
> "[In SmallDraw] the controller is independent of command key processing and additional keys may be added to the model without changing the controller's method."

The SmallDrawController is responsible for all input, and can now check for keyboard activity in its normal control sequence. All of the operations listed above are performed by the SmallDraw model. When the controller senses that a command key has been pressed, it forwards the key to the model for processing. This way, the controller is independent of command key processing and additional keys may be added to the model without changing the controller's method. The SmallDraw instance method that processes command keys looks very much like the list of operations above:

```
processCommandKey: aKey
    "Respond to aKey which may correspond to one of the receiver's
        menu commands. If not, ignore it."
    aKey = Character backspace ifTrue: [self delete].
    aKey = $x ifTrue: [self cut].
    aKey = $c ifTrue: [self copy].
    aKey = $v ifTrue: [self paste].
    aKey = $f ifTrue: [self moveForward].
    aKey = $j ifTrue: [self moveBackward].
    aKey = $d ifTrue: [self duplicate].
    aKey = $a ifTrue: [self selectAll].
    aKey = $k ifTrue: [self doAlignment].
    aKey = $g ifTrue: [self group].
    aKey = $G ifTrue: [self unGroup].
```

SmallDraw menus are modified to indicate the keyboard commands that may substitute for menu operations (see Figure 3):

SmallDrawController is only slightly modified in order to handle keyboard events. One method is added to detect and process any keyboard activity:




```
processKeyboard
    "Determine whether the user pressed the keyboard. If so, read the
        key and pass it on to the model."
    self sensor keyboardPressed ifTrue: [| keyHit |
        KeyHit := self sensor keyboardEvent keyValue.
        "Check for backspace here."
        keyHit = Character backspace ifTrue:  ,
            [self model processCommandKey: keyHit].
        (self sensor altDown or: [self sensor metaDown]) ifTrue: [
        "KeyValues are lowercase so we must convert to uppercase if the
                shift key is down."
            self sensor shiftDown ifTrue:
                [keyHit := keyHit asUppercase].
            self model processCommandKey: keyHit]]
```

and one inherited method is overwritten to include the keyboard method in its control loop:

```
controlActivity
    "First check the keyboard and then do the usual."
    self processKeyboard.
    super controlActivity.
```

SCROLLING THE VIEW

SmallDrawView can become a scrollable view by defining it as a subclass of ScrollingView. The class comments for ScrollingView include the following information:

```
Subclasses must implement the following messages:
    accessing
        displayObject
    scrolling
        scrollBy:
        scrollHorizontally:
        scrollVertically:
```

DisplayObject must be able to respond to the message bounds. DisplayObject is the object being scrolled in the view, in this case the SmallDraw drawing. SmallDrawView needs to know how big the SmallDraw document is so that the scroll bars can be properly scaled. SmallDraw's new instance variable, pages, is an instance of a Point that defines the number of pages lined up horizontally and vertically. The minimum is 1@1, or one page. For two pages side by side, pages would be 2@1, and so on. The document automatically increases in pages if objects are translated or scaled such that they extend beyond the rightmost or bottommost pages of the document. The SmallDrawController ensures that objects are not allowed to extend beyond the leftmost or topmost pages.

The size of the document is obtained by asking SmallDraw for its bounds:

```
bounds
    ^0@0 extent: self documentSize
```

where the page configuration is converted to pixels by multiplying an 8 1/2 x 11 inch sheet of paper (assuming 1/2 inch margins all around) by the number of pixels per inch:

```
documentSize
    "Answer the size of the document in terms of the number of 8.5 x
```

```
    11 inch pages."
    ^self pages * self pageSizeInPixels

pageSizeInPixels
    "Answer the size of one 8.5 x 11 inch page (with 1/2 inch margins),
        scaled by the number of pixels per inch (72). This number is
        calculated as: ((7.5@10) * 72) rounded."
    ^540@720
```

To ensure proper scaling of the scrolled object, SmallDrawView defines the following method:

```
dataExtent
    ^self displayObject bounds extent * self displayScale
```

Scroll bars rely on a scrolling grid in which the inherited value for scrollGrid is 1@1. Using pasteOffset, SmallDrawView can be defined so that scrolling occurs in larger intervals. SmallDrawView provides a menu option to turn the grid on or off and SmallDrawController uses its view's grid for selecting points in the view.

Opening SmallDraw with a scrolling view is done as before by placing the SmallDrawView in an EdgeWidgetWrapper but now a horizontal scroll bar is also included (see Figure 4):

```
openScrolling
    "SmallDraw new openScrolling"
    ScheduledWindow new
        label: 'SmallDraw';
        component: (EdgeWidgetWrapper on:
            (SmallDrawView model: self)) useHorizontalScrollBar;
        openWithExtent: 200@200
```

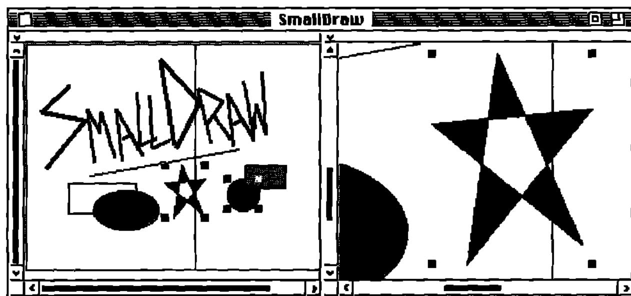


Figure 4. Two scrolling views (25% and 100%) and two pages side by side.

SUMMARY

Building on the first two SmallDraw articles, this final article has presented further enhancements to SmallDraw to demonstrate Release 4 graphics and MVC application construction. Though far from perfect, it should give beginners a good start on their own development.

Certainly many improvements and enhancements can be made to SmallDraw. New types of graphic objects, such as Text, Images, and Bezier curves (included in Release 4.1), can be added. Other object operations can be defined, such as rotation, smoothing of polygons, editing individual points on a polygon, undo, or auto scrolling of the drawing while translating or scaling objects beyond the extent of the view. Advanced functionality can be provided to allow for saving drawings to files, PostScript or LaTeX printing of the drawing (e.g., a GraphicsContext subclass that outputs PostScript), or sharing of graphic objects with other Smalltalk applications.

The complete source code corresponding to each of the three SmallDraw articles can be obtained from the University of Illinois and Manchester archives. They are identified as SmallDraw1, SmallDraw2, and SmallDraw3. The source code is available to all with no restrictions. I ask only that proper credit be given so that I may hear from those who have benefited. I also encourage those who make improvements or additions to SmallDraw to make them available through the archives for others' education and use. ■

Dan Benson completed his PhD in Electrical Engineering at the University of Washington where he developed a 3-D spatial database for human anatomy using Smalltalk and the GemStone ODBMS. He is now a Research Scientist with Siemens working in the area of Image Management and Distribution. He may be contacted at: Siemens Corporate Research, Inc., 755 College Road East, Princeton, NJ 08540, or by email: benson@siemens.siemens.com.

VOSS

Virtual Object Storage System for

Smalltalk/V

Seamless persistent object management for all Smalltalk/V applications

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

Some comments we have received about VOSS:

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

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

Logic ARTS

VOSS/286 $595 (Personal $199), VOSS/Windows $750 (Personal $299) (Personal versions exclude items marked (np)).
Quantity discounts from 30% for two or more copies. (Ask for details)
Visa, MasterCard and EuroCard accepted. Please add $15 for shipping.
Logic Arts Ltd 75 Hemingford Road, Cambridge, England, CB1 3BY
TEL: +44 223 212392 FAX: +44 223 245171

TO SUBSCRIBE TO

The Smalltalk Report,

CALL 212/274-0640 OR

FAX YOUR REQUEST TO 212/274-0646

Alan Knight

What *else* is wrong with OOP?

This might more accurately be called "What else do people on USENET think is wrong with OOP?" While there are certainly areas in which OOP could be improved, there are many misconceptions and false criticisms—so many, in fact, that I ran out of space for them last month and am continuing the topic here.

Let's start with one of the most common complaints: application areas for which OOP is inappropriate.

OOP CAN'T HANDLE PROBLEMS LIKE...

Harry Erwin (erwin@trwacs.fp.trw.com) writes:

> **OOP can be a disadvantage if the problem domain does not lend itself conveniently to object representations. For example, many algorithms consist of a primary control loop operating on passive things, and a Pascal or Ada program of the traditional mode is more efficient and clearer.**

If true, this represents a severe restriction of the OOP domain. Many algorithms fit the pattern of a loop operating on passive things; if OOP can't handle them, most programming is ruled out. Objects will have to be relegated to simple GUI tasks, error handling, and other algorithmically trivial areas.

In my opinion, it is not difficult to describe many algorithms in terms of a main loop. The loop can be written as:

```
aBunchOfPassiveThings do: [:passiveThing |
    algorithmManager process: passiveThing].
```

The code gets more complicated if we include initialization and post-processing code, or if it has to use a more complex method of choosing the next item, but I do not think a Pascal or Ada program could be clearer.

The complicated part is the processing of each "passive thing," which usually consists of elaborate manipulations of various data structures. The algorithms literature considers it good form to describe these manipulations in terms of operations on abstract data types. OOP usually handles abstract data types very well, so it is actually very good for this kind of work.

BUT THAT'S NOT REALLY OBJECT ORIENTED

I'm quite happy with the general method of writing "traditional" algorithms using OOP because (1) the program structures correspond well with typical algorithm description, (2) there's good potential for reuse of abstract data type classes, (3) it's clearly suitable for implementation in an OO language, and (4) it nicely groups together the algorithm data in the AlgorithmXManager class.

A recurring theme among complaints about OOP is that it is "not really object-oriented." But OOP solutions to problems are often rejected as not being faithful to the principles of object orientation because of a misguided idea of what objects are about.

THE PRINCIPLES OF OOP

What does it mean for a solution to be object-oriented? On what basis are these kinds of solutions rejected? Are these ideas valid and, if so, are they important enough to make us discard good solutions?

The standard definition of an OO language says that it should support encapsulation, polymorphism, and inheritance. True, but these are language features, not a set of guiding principles. The dictionary is even less helpful. Mine traces the word *object* to the Latin *objectum,* literally meaning "something thrown before or against." Its roots are the words *ob* (against) and *jacio* (to throw). Since we are interested in perceptions of OOP, let's find out what people on USENET think.

David Myers (dem@meaddata.com) writes:

> **Once people learn Object-Oriented Design, they seem to fall into two schools of thought. I'm interested in your thoughts on which, if either, is more correct.**
> **The first camp I'll call "Strict OOD." They believe that all functions that need to modify some object must necessarily be member functions of that object....**
> **The second camp I'll call "Reality OOD." They don't believe in taking things as far as the first camp if the resulting model wouldn't fit with their perception of reality....The Reality OOD folks want to build an OO system so that its components closely represent the world they are trying to model....**

and later expands:

> **You want to model a cow, and want to get milk from the cow and put it in a vat...Strict OOD might say, "Just add a method 'Cow, milk yourself,' which puts the milk right in the vat. Leave the details to the cow." Obviously, Reality OOD would say something different. "Cow, present ud-**

ders. Udders are the interface here, and we can 'pass' a cow to a farmer object to get the cow milked and the milk in the vat. The farmer contains the knowledge of how milking should be done, not the cow."

...say we now have a better way to milk a cow, with a milking machine. Strict OOD would say, "Modify the cow to understand how to use the milking machine..." Reality OOD would say, "Just 'pass' the cow to the new machine. The cow doesn't need to change as it already provides the necessary interface."

...Another example. Say you have some glob of data, and you want to run N validation processes against it...Where do these processes go? Strict OOD, "Part of the glob, obviously. That's what they act upon." Reality OOD, "They're separate from the glob, and use whatever interface is provided by the glob to do their work."

This is quite interesting, because it's a well-considered, thoughtful posting based fundamentally on false ideas of OOP. It arises from the basic question of where to put methods, but in my opinion gets the principles wrong. I see the method placement question as a conflict between the principles of coupling and cohesion.

Consider the validation example, which expresses this most clearly. A Validator class is a good idea. It groups related methods (for testing) together, and removes clutter from the class being tested. It's easy to add additional validation checks, and seems to be the only method that generalizes to consistency checks involving several different objects.

On the other hand, we should hide internal representations to minimize coupling. The internals of a class should not be exposed, and we expect validation to require access to these details at least some of the time.

A good compromise is to use both techniques. Use class methods to implement tests that depend on internal representations, preferably using a consistent naming scheme. Tests that can be done through the public interface should be implemented through a Validator class, which when validating can also invoke the appropriate self-testing methods in the individual class.

The above posting is based on two false ideas, one in each camp. Mr. Myers presents "Strict OOD" as the orthodoxy of the OOP gurus. It dictates that any method modifying an object's state must belong to the class of that object. On the surface this sounds reasonable, very much like encapsulation, but it's an overgeneralization that simply cannot work in practice.

Encapsulation restricts the set of methods that can access an object's internal representation to those in its class. This is enforced in Smalltalk, but it is possible to short-circuit the restriction by writing get/set methods for each instance variable. A method that accesses an object's state through message sends could be placed anywhere, but if it operates primarily on one object it is good style to make it a method in that class.

There's a big difference, however, between good style and an enforced rule. In particular, the "strict" position does not allow the possibility of methods that modify (or even access) more than one object. This disallows such a simple thing as a bank transaction, where one account is incremented and another decremented.

The "Reality OOD" camp allows such methods, but then runs back into the question of method placement, as K. Srinivasan (srini@gtsurya.gatech.edu) points out:

I am interested in developing OO models to represent manufacturing enterprises. I ran into the very same problem you've described — A method "process a part" seems to alter the states of the part object, the machine object and the operator object, and hence is a candidate for being a method belonging to any of them. To make it a method of one, say "part," and make that object a client of other two objects (operator and machine) will work. However, it seems to be a highly arbitrary decision.

I agree wholeheartedly. If two or more things interact, and the states are all changing, then the decision to place a method handling this interaction is arbitrary. If the interaction is sufficiently important, it may be worthwhile modeling it as an object itself. Ralph Johnson (johnson@cs.uiuc.edu) discusses this in the context of the milking example.

The real issue is how to divide responsibilities among objects.... Why not give the vat responsibility for taking the milk from a cow? Without knowing anything about the real world domain and what is likely to change, any of three possibilities is just as likely. We have a transaction between object C and object V, and the question is whether we should introduce a new object F to model the transaction (transactor) or we should make the transaction a method of C or V. In general, it all depends!...If we have a simple system whether nothing changes, then it might make sense to put the responsibility for the transaction in C. If we knew that the transaction itself was never going to change, and that C was, (i.e. we want to milk sheep, goats, horses, yaks, etc.) then it might be better to put it in V. If the transaction itself is going to change (i.e. use a milking machine) then it would be better to make it an object.

Once again we hear the cry that this solution is "not really object-oriented," which brings us to the second, and more important, fallacy.

OOP AND THE REAL WORLD

Choosing the right name for something is important. A name should be short, easy to remember, and clearly communicate the essential idea. Unfortunately, "object-oriented" fails in the last category.

The problem is that everyone knows what an object is. We intuitively "know" that object-oriented programming is all about objects: concrete, physical things that we can, with enough machinery, pick up and throw. Processes can't be objects. Relationships can't be objects. Concepts can't be objects. OOP is "good" because it writes programs that perfectly mimic the real world, and an OO program is "good" in direct proportion to its mimicry—like neural networks, which we all know

work just like human brains. Being told that OOP is good for simulation and that it naturally models the problem domain only makes these misconceptions worse.

Smalltalk programmers tend to transcend these ideas more quickly than others because they're confronted with examples of Schedulers, Controllers, Associations, and other non-concrete classes. Even so, the misconceptions are very widespread. Let's look at some concrete examples.

Objects are always concrete nouns

Dan Weinreb (dlw@odi.com) writes:

> **This topic comes up again and again whenever semantic data modeling is being discussed. I've seen it in papers from over ten years ago. After reading a bunch of the literature in this area I have come to the conclusion that there doesn't seem to be any completely satisfying answer. Either you end up having these objects that only model relationships rather than modeling "things" in the problem domain, or else you end up inventing constructs that are annoyingly complex and often disturbingly similar to objects themselves.**

and Doug MacDonald (doug@softwords.bc.ca) writes:

> **This thread raises what I have always considered to be a shortcoming of OO scheme of modeling the world: while it allows us to capture complex classifications and instances, it does NOT provide the idea of relationships among objects. Yes, we can "send messages" among objects, provide well-structured access functions. But this does not address the central problem. We end up with forced concepts like relationship classes to deal with the cow-milk type puzzles.**

This literal interpretation of objects corresponding only to physical "things" is probably the single most prevalent misconception about OOP. It is the main reason people reject solutions that include an AlgorithmManager or a class representing the relationship between cows and farmers. I've seen many other examples, including database discussions that assumed an ODBMS could model only physical things, and that an RDBMS could only model relationships. In a similarly literal vein, I've seen C described as a functional language because it has functions.

Naturally, there are many who do not share these beliefs. Eric Smith (eric@tfs.com) writes:

> **There is nothing "forced" about relationship classes. Relationships are objects, period. The word "relationship" is a noun. A relationship object should contain references to its target objects, functions to return information about its target objects and about various aspects of the relationship between them, and functions to modify the relationship.**

Mike Wirth (mcw@cs.rice.edu) writes:

> **Nothing unnatural about it at all. Associations between objects are every bit as much "real world" objects as the objects being associated. Ask your spouse or "significant other."**

And Ralph Johnson (johnson@cs.uiuc.edu), who seems to have encountered these ideas before, anticipated the objections in the same posting quoted above:

> **There is NOTHING wrong with having objects that represent processes. It is true that novice OO designers make a lot of such objects that are bad design, but good OO designers make those kinds of objects, too. You just need to have a good reason for introducing a new object.**

The fundamental point of OOP is abstraction. A good OOP design should correspond to ideas in the problem domain. Whether those happen to be ideas about things that can be touched or about relationships, processes, or concepts is irrelevant. One of the best metrics for this is naming. If someone familiar with the domain can look at a class name and immediately have some idea what it does, then it's probably a good class for that domain.

There is exactly one "right" OOP design for a problem

Given that the objective is a perfect model of reality, then all OO designs should converge. After all, there's only one real world. This results in much disappointment when people discover that OOP, like any other kind of programming, still has design decisions and trade-offs.

David A. Hasan (hasan@ut-emx.uucp) writes:

> **...the "map" between OO methods/objects and what is going on in the real world is NOT unique. There can be different interpretations on which objects should carry out which methods based on how the real world activities are "best modeled." Therefore a choice must be made in specifying object interfaces, and making this choice might unduly constrain future versions of the system...**

This is entirely true, but it is based on vastly inflated expectations of what OOP can do.

bobm@Ingres.COM (Bob McQueer) replies:

> **What problem you are trying to solve defines "proper," I think. I can see us having the same problems we have always had when trying to "grow" new functionality into a design that didn't anticipate growth in that direction. Note that expediency will dictate that you can't make provisions for EVERY possible direction of growth, also as it always has.... I think what I'm saying is that while the OO paradigm is a useful tool, you can't expect the existence of a paradigm to do all your work for you. There is NOT a unique map, and it takes proper use of the tool to define the map which serves your purposes.**

THE REAL WORLD AGAIN

The idea of modeling the real world in detail is fallacious. In what we call "reality," most things are human-imposed concepts. Reality consists mostly of interactions between elementary particles; the higher-level structures we perceive are ab-

continued on page 22...

Juanita Ewing

Extending the Collection hierarchy

In my last column, I discussed creating subclasses and two heuristics for selecting superclasses. This month I will continue the discussion on subclassing with a case study that extends the Collection hierarchy. We will create a new Collection class that contains unique elements and also maintains the order of these elements.

HEURISTICS REVIEW

A key step in creating a new subclass is to select a suitable superclass. The heuristics for selecting a superclass are:

Heuristic One: Look for a class that fits the is-kind-of or is-type-of relationship with your new subclass.

Heuristic Two: Look for a class with behavior that is similar to the desired behavior of the new subclass.

CASE STUDY

We want to create a new data structure class that holds elements in order and disallows duplicate elements. When sent a request to add a duplicate object, the request should be quietly ignored.

This new data structure class contains elements similar to Arrays, Strings and other Collection subclasses. Because of these similarities, we will begin our search for candidate superclasses in the Collection hierarchy. Two classes immediately stand out:

- OrderedCollections keep elements in order.
- Sets store each element only once, disallowing duplicate elements.

The combination of these characteristics is what we want for our new class. A good descriptive name for our new class is OrderedSet.

APPLY HEURISTICS

Where should we insert our new class, OrderedSet, into the hierarchy? Our first heuristic is to look for potential superclasses that match the is-kind-of criteria. We use is-kind-of as a shorthand for categorization based on characteristics. The significant characteristics and their classes used in this determination are:

- vary number of elements (Collection)
- store arbitrary objects (Collection)
- dynamically add and remove elements (Collection)
- enumerate (Collection)
- store elements in order (OrderedCollection)
- store unique elements (Set)

The desired characteristics of OrderedSet are closest to those of OrderedCollection and Set, so OrderedSet could be a-kind-of Set or a-kind-of OrderedCollection.

In a system that supports multiple inheritance, we might be tempted to have two superclasses, Set and OrderedCollection. In Smalltalk we must choose a single superclass, either Set or OrderedCollection.

Our second heuristic is to choose candidate superclasses with suitable public behavior. Let's compare the candidate classes we've selected, Set and OrderedCollection, in terms of behavior. Set and OrderedCollection have a common superclass, Collection, so we can ignore public behavior from the Collection on up.

If we were to make OrderedSet a subclass of Set, it would inherit these methods from Set:

```
add:
do:
includes:
occurencesOf:
remove:ifAbsent:
size
```

All of these methods also have an implementation in the abstract superclass Collection, so Set doesn't add any new public behavior to the behavior from the common superclass.

If OrderedSet were a subclass of OrderedCollection, it would inherit behavior from OrderedCollection and IndexedCollection (or OrderedCollection and SequencableCollection in Objectworks\Smalltalk). OrderedCollection has adding and removing methods and many more methods related to its element-ordering characteristic. The list of methods includes:

```
add:                 addfirst:
add:after:           addLast:
add:afterIndex:      remove:ifAbsent:
add:before:          removeFirst
add:beforeIndex:     removeLast
```

Many of these methods are extensions of the public behavior from the common superclass Collection.

The public behaviors for Sets and OrderedCollections have some similarities. In fact, the behavior of Set is a subset of the behavior of OrderedCollection, which makes Set the behavioral supertype of OrderedCollection. Set doesn't add any additional

behavior, so we just need to determine whether the additional behavior in OrderedCollection is desirable.

Because instances of OrderedSet maintain elements in order, we will need public behavior to support the ordering characteristic. The behavior in OrderedCollection is a good set of behavior for supporting this characteristic. In addition, if the behavior of OrderedSet is the same as for OrderedCollection, the interchangeability of the classes is better and therefore the classes are easier to reuse. Based on behavioral analysis, the best superclass for OrderedSet is OrderedCollection.

IMPLEMENTATION

We can also look in more detail at what is required to implement OrderedSet. The implementation of OrderedCollection uses an indexable portion or indexable object, as well as instance variables to keep track of valid indices. Set is implemented with hashing for efficiency in determining uniqueness of elements. If a Set already contains an element, it quietly ignores the request to add an element.

OrderedSet needs to support instances with a large number of elements. Hashing the elements is a good way to support large numbers. OrderedCollections would potentially have to examine every element before determining if the addition of an element would be a duplication. To maintain order and enforce uniqueness we will use two structures, one to implement the unique elements characteristic, and one to implement the ordering characteristic, as shown in Figure 1.

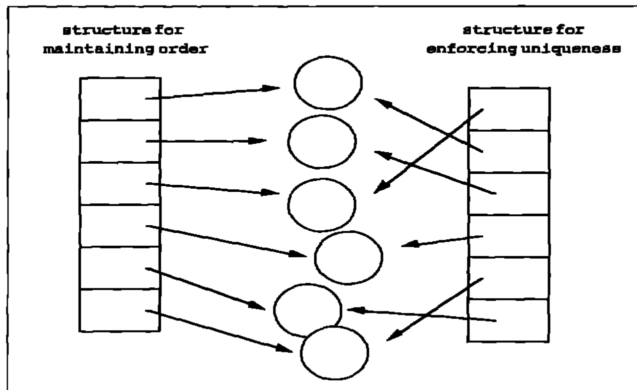


Figure 1. Using multiple structures.

Now we will examine the implementations with each of our candidate superclasses. If OrderedSet is a subclass of OrderedCollection, we inherit the portion that stores elements in order and we need to implement the portion that hashes and enforces uniqueness. The structure and behavior for maintaining order is inherited from OrderedCollection, and the structure for enforcing uniqueness can be stored in an instance variable. This structure could be an instance of Set.

With this alternative, some inherited methods would need to be overridden. All the add and remove methods must potentially be altered to maintain both structures. As seen in the list of public behavior, there are a number of these methods, such as add:, add:after:, add:afterIndex:, addfirst:, removefirst and removeLast. Fortunately, not all these methods have to be overridden because some of them call each other. We would want to override includes: because the hashing used in the uniqueness structure gives us a quick lookup of elements. We would not override do: because it operates on the inherited structure that maintains order.

If OrderedSet were a subclass of Set, the inherited structure is the one that enforces uniqueness; an auxiliary structure for maintaining order is referenced from an instance variable. Presumably, the order maintaining structure would be an instance of OrderedCollection.

We would also need to override adding and removing methods—there is just one of each. The majority of coding is in implementing behavior that implements the element ordering characteristic. We would not need to override includes: because we inherit the version that makes use of hashing, but we would need to override do: so that we process elements in the ordered defined by the order maintaining structure.

NAMING

Other criteria that might bias our judgment are implications of a class's name. If a class hierarchy is part of the public interface for a library, it might be easier for users to locate a class located in a logical place in the hierarchy. With a class called OrderedSet, users are more likely to look for this class as a specialization of Set. They might not find it as easily if it is a subclass of OrderedCollection.

CONCLUSION

We make OrderedSet a subclass of OrderedCollection because:

- The behavior of OrderedCollection is more suitable than the behavior of Set.
- It is more likely that the behavior will be interchangeable if the relationship between the two classes is explicit.
- There are fewer methods, overridden and new, that must be implemented in OrderedSet.

Furthermore, by browsing the Collection hierarchy, developers will generally examine several Collection classes at a time, and will probably notice OrderedSet as a subclass of OrderedCollection.

The is-kind-of heuristic is useful for generating candidate superclasses. Its intuitive nature can be an advantage. However, analysis of public behavior often yields a better selection. If we only used the is-kind-of heuristic in our case study, we would be most likely to make OrderedSet a subclass of Set. On the other hand, when we use the public behavior heuristic, we conclude that OrderedCollection is a better choice. ■

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

Kent Beck

ValueModel idioms

My last column outlined ways of using dependency as embodied in Smalltalk's update and changed messages. ParcPlace's release 4 of Objectworks\Smalltalk introduced a significant refinement of dependency called ValueModel which addresses some of the shortcomings of the classic style of dependency management.

CLASSIC SMALLTALK STYLE

Here is another example of the classic style of Smalltalk change propagation. A Mandelbrot renders a portion of the Mandelbrot set while it measures performance.

```
Mandelbrot
    superclass: Model
    instance variables: region flops
```

A Mandelbrot object renders the portion of the Mandelbrot set in region (a Rectangle with floating point coordinates) on an Image when sent displayOn:. Assume we have implemented a primitive rendering method that returns the number of floating point operations it initiates as it displays. The DisplayOn: method divides the number of operations by the rendering time to compute the number of floating point operations per second, which will be stored in flops.

```
displayOn: anImage
    | time ops |
    time := Time millisecondsToRun:
        [ops := self primDisplayOn:anImage].
    self flops: ops / time / 1000
```

The model responds to openflops by creating a window that displays the value of flops.

```
openflops
    | window |
    window := ScheduledWindow new.
    window addChild: (TextView on: self aspect: #flopsString
        change:nil menu: nil)
window open
```

Some users complain that putting an open method in the model allows too much of the interface to leak through. But in my opinion one is free to open any kind of window, and if the model offers a default way, so much the better. Putting open in the model keeps the code together; if more flexibility is needed later it can always be moved.

TextView's symbol flopsString is used by the view both to recognize an interesting broadcast and as a message to the model to return a string suitable for viewing. The model thus needs to respond to flopsString.

```
flopsString
    ^self flops printString, 'flops'
```

Now all that remains to update the view is to propagate a change whenever the flops change.

```
flops: aNumber
    flops := aNumber.
    self changed: #flopsString
```

Already the interface is beginning to leak into the model. Because the example interface uses the symbol #flopsString, the model must have this particular symbol built in. Other interfaces viewing other aspects of the model dependent on the measured flops will require additional broadcasts when the flops change. The model is no longer insulated from changes to the interface.

Let's refine the model a bit to see where this style of change propagation begins to fall apart. What if instead of displaying the last value of flops we want to display the average of recent values? flops holds an OrderedCollection instead of a Number.

```
initialize
    flops := OrderedCollection new
```

The setting method adds to the collection instead of changing the instance variable.

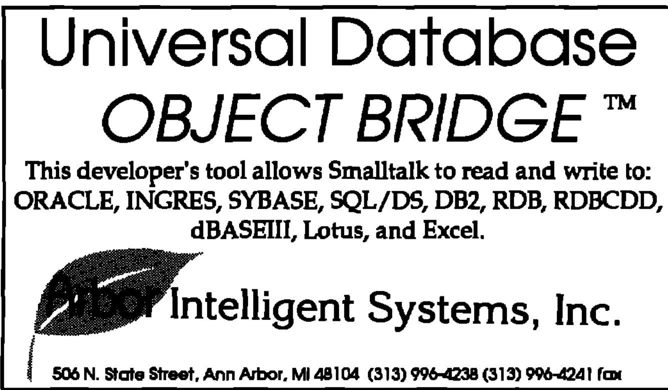
```
flops: aNumber
    flops addLast: aNumber.
    self changed: #flopsString
```

The accessing method has to compute the average instead of just returning the value.

```
flops
    flops isEmpty ifTrue: [^float zero].
    ^(flops inject: float zero into: [:sum :each | sum + each])
        / flops size
```

The above code is still fairly clean from an implementation perspective. From a design standpoint, though, it is a dangerous path.

The first problem is that the needs of the interface influence our implementation of the model. Conversely, our concept of an interface is constrained by the way we have implemented the model. The separation of model from interface, supported at the implementation level by broadcasting changes, merely reappears as a design problem. In other words, the letter of



"separate model and interface" is satisfied because the model makes no direct reference to the interface, but the spirit is violated because interface decisions have caused us to change a model that should be oblivious to such concerns.

Other views with other aspects require inserting more hardwired broadcast messages. In large projects, this process of broadcast accretion leads to a bewildering profusion of broadcasts, often with intricate time dependencies.

Another problem is that this style of programming discourages reuse. Each instance variable is a special case, to be handled by special case code. For example, suppose we are working in a multiprocessor environment and want to view a running average of the number of processors active during rendering. We could add an instance variable, utilization, with accessing and setting methods that are copies of the respective messages for flops, but we could do no better at reuse than copy and paste.

This last point suggests that state change and change propagation somehow must be folded together into a new object. This object will be used instead of a bare instance variable as a model for views. We can create a family of these objects to model the different ways of viewing state changes over time. By using various kinds of objects in varying circumstances we can change the interaction supported by the model without changing the model itself.

The most common solution to these problems is to separate the model into a "browser" object and a clean underlying model without broadcasts (see Figure 1). The browser mediates between the user interface and the "real" model, translating user requests into messages to the model and propagating changes back to the interface. Although fairly simple conceptually, this style of programming introduces another layer of objects between the user and the model without addressing the problem of multiple browsers on the same model (for example, the problem of updating the source code of a method appearing in more than one Browser).

VALUE MODEL STYLE

ValueModels in Objectworks\Smalltalk Release 4 fill the role of an interaction model. Rather than appearing between the domain model and the interface, ValueModels are placed "beneath" the domain model. This allows the view to interact directly with the state of the domain model and does not clutter the model itself with interaction concerns.

Here's how an ideal implementation can be applied to our example:

```
ValueModel
    superclass: Model
    instance variables: value

value
    ^value

value: anObject
    value := anObject
    self changed: #value
```

We can recast Mandelbrot to use this simple ValueModel. First, the initialization method sets flops to a ValueModel.

```
initialize
    flops := ValueModel new
```

When accessing or setting the value you must remember to send messages to flops and not just use the instance variable. Religious use of accessing and setting methods, though, can hide this detail from the rest of the object.

```
flops
    ^flops value
```

Note that when the value is set the Mandelbrot no longer needs to propagate changes.

```
flops: aNumber
    flops value: aNumber
```

When making a view to display flops the ValueModel is the model of the TextView, not the Mandelbrot.

```
openflops
    | window |
    window := ScheduledWindow new.
    window addChild: (TextView on: flops aspect: #value
        change: nil menu: nil).
    window open
```

We now have a system with the same functionality as the simplest one described above. Figure 2 diagrams the relationships between the various components in the value model-style Mandelbrot.

The worth of ValueModels becomes apparent when we display a running average rather than a single value. The change is made creating a subclass of ValueModel called AveragingValueModel, which accumulates a history of values in response to value:messages.

```
AveragingValueModel
    superclass: ValueModel
    instance variables: none
```

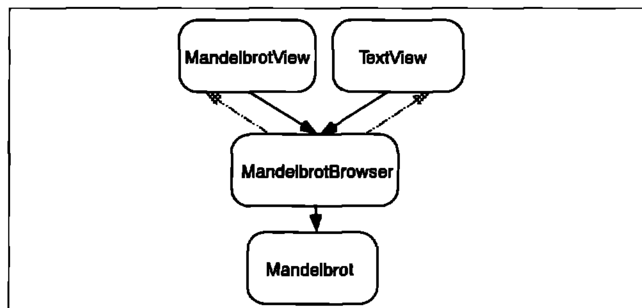


Figure 1. Classic separation of model and interface.

```
initialize
    value := OrderedCollection new
value
    value isEmpty ifTrue: [^Float zero].
    ^(value inject: Float zero into: [:sum :each | sum + each])
        / value size
value: anObject
    value addLast: anObject
```

We can install the new behavior by changing Mandelbrot>>initialize.

```
initialize
    flops := AveragingValueModel new
```

No other changes to the model are necessary. When we want to open a window on a running average of processor utilization we can create another AveragingValueModel. We do not need to duplicate any code.

The model has acquired a large measure of independence from changes mandated by the interface. For many interface changes we no longer need to touch code in the domain model beyond modifying the initialization. We instantiate a new kind of ValueModel and the rest of the model remains unchanged.

THE REST OF THE STORY

The above code still doesn't quite work. The TextView expects a String or a Text from its model, and the ValueModel in this case returns a Number. The release 4.1 solution is to interpose another object, called a PluggableAdaptor, between the model and the view. A PluggableAdaptor contains three blocks. The first is invoked when it receives the message value. The block takes one argument, the adaptor's model (in this case the ValueModel), and by default returns the result of sending value to the model. The block can be used to arbitrarily transform the value. In our case we want to create a string from the number:

```
openflops
    | window adaptor |
    window := ScheduledWindow new.
    adaptor := AspectAdaptor on: flops.
    adaptor getBlock: [:m | m value printString, ' flops'].
    window addChild: (TextView on: adaptor aspect: #value
        change: nil menu: nil).
    window open
```

The second block in a PluggableAdaptor is evaluated when the adaptor receives the value: message. The block is invoked with the model and the new value as arguments. By default it passes the message along to the model. This block translates the value from a form the view understands to one the model

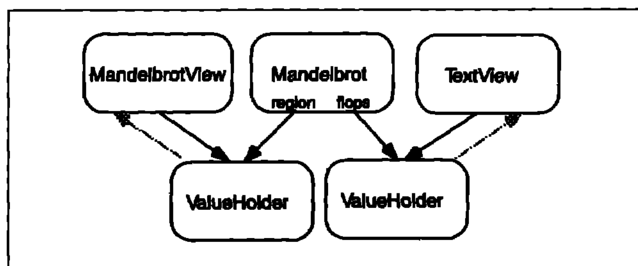


Figure 2. ValueHolder style separation of model and interface.

Zippy Object Oriented Memory™
Object Oriented Database Management System
The ONLY ODBMS for Smalltalk for under $1000 that delivers Persistent Object Storage on Disk via a Zippy B+Tree Database Retrieval Engine!
for Smalltalk/V and Smalltalk-80
All Platforms $199.95
Source Code Included
Hierarchical Applications Limited
ZOOM
(512) 837-2117
12407 Mopac Expwy N., Suite #100-266
Austin, TX 78758

understands. If it was possible to change the flops rating, we might write something like this:

```
openflops
    | window adaptor |
    window := ScheduledWindow new.
    adaptor := AspectAdaptor on: flops.
    adaptor getBlock: [:m | m value printString, 'flops'].
    adaptor putBlock: [:m :v |
        m value: (Number readFrom: v readStream)].
    window addChild: (TextView on: adaptor aspect: #value
        change: nil menu: nil).
    window open
```

The final PluggableAdaptor block is used to filter update messages. The block takes three arguments: the model, the aspect from the update: message, and the optional parameter from the update: message. The block evaluates to a boolean that is used to decide whether or not to forward the update. In our example we may not want to update the text if the flops rating is too low. We could change openflops as follows:

```
openflops
    | window adaptor |
    window := ScheduledWindow new.
    adaptor := AspectAdaptor on: flops.
    adaptor getBlock: [:m | m value printString, ' flops'].
    adaptor putBlock: [:m :v | m value: (Number readFrom: v
        readStream)].
    adaptor updateBlock: [:m :a :p | m value > 1e6].
    window addChild: (TextView on: adaptor aspect: #value
        change: nil menu: nil).
    window open
```

When an object is dependent on two or more ValueModels it is often important to distinguish which one is generating the broadcast message. One solution is to take advantage of the full generality of the update message:

A cleaner solution is to use the update block of a pluggable adaptor to generate different updates for each ValueModel. The initialization would look like this:

```
initializeWith: model1 with: model2
    | adaptor1 adaptor2 |
    adaptor1 := PluggableAdaptor on: model1.
    adaptor1 updateBlock: [:m :v :p | v == #value
        ifTrue: [adaptor1 changed: #value1]].
    adaptor1 addDependent: self.
    adaptor2 := PluggableAdaptor on: model2.
    adaptor2 updateBlock: [:m :v :p | v == #value
        ifTrue: [adaptor1changed: #value2]].
    adaptor2 addDependent: self
```

Then the update method can look like this:

```
update: aSymbol
    aSymbol == #value1 ifTrue: [self updateValue1].
    aSymbol == #value2 ifTrue: [self updateValue2]
```

The preceding information is written assuming ValueModel holds values. In the real system, though, ValueModel is an abstract superclass, and the subclass acting as ValueModel above is really called ValueHolder. PluggableAdaptor is also a subclass of ValueModel. Other subclasses (like AveragingValueModel) should arise as the full utility of the ValueModel style becomes apparent.

LAZY VIEWS

A final idiom that accompanies Objectworks\Smalltalk release 4 and later is lazy updating of views. Back when dinosaurs ruled the earth and Smalltalk did its own window management, it was common to directly redisplay a view in response to an update:

```
update: aSymbol
    (self interestedIn: aSymbol) ifTrue: [self displayView]
```

A serious problem with this strategy is that the view will be redisplayed several times if multiple update messages come in. Multiple updates look bad and slow your programs down. This is especially true with the expanded use of broadcast messages in release 4.

When you implement views in release 4 and later, you should never directly redisplay the view. Instead the view should send itself an invalidate message:

```
update: aSymbol
    (self interestedIn: aSymbol) ifTrue: [self invalidate]
```

These invalidations are pooled together. The next time a Controller sends itself poll (or someone explicitly sends checkForEvents to ScheduledControllers) all views with some invalid area will be asked to display. This ensures that if there is a change to a model causing several views to update they will redisplay as simultaneously as possible.

CONCLUSION

The ValueModel style of coding manages complexity by strictly separating interface and model.

We have just begun to explore the range of possibilities inherent in the ValueModel style. You can expect to discover new uses as you begin using it yourself. If you find new ValueModels, or new uses for the existing ones, please drop me a line so I can publish them here. ■

Kent Beck has been discovering Smalltalk idioms for eight years at Tektronix, Apple Computer, and MasPars Computer. He is also the founder of First Class Software, which develops and distributes reengineering products for Smalltalk. He can be reached at First Class Software, P.O. Box 226, Boulder Creek, CA 95006-0226

THE BEST OF...*continued from page 16*

stractions useful in some specific domains. Reality can have very poor software engineering principles.

Jeff Alger (alger@applelink.apple.com) writes:

Seldom are you ever modeling the real world in software. The real world is the problem; why would you want to just simulate it? Objects and classes in a piece of software are nothing more than metaphors. In fact, direct simulations of real-world objects lead to very poor object-oriented architectures with little or no modularity and that are highly unstable. Early on one learns that a Paycheck object should print itself and a Block object should move itself around on a screen. This is not the real world.

And Philip Santas (santas@inf.ethz.ch) points out:

There is no such thing as information hiding in the real world.

CONCLUSIONS

Since this column has been devoted to what's *wrong* with OOP, I ought to conclude with what I think is right:

1. OOP is not a panacea. OOP is good for improving reuse; it does not make reuse automatic. If I write a Car class for modeling traffic flow and you write a Car class for modeling the physics of collisions, our chances of being able to use the same class are small. Programs should carefully choose what they're trying to model.
2. Don't try to model the real world in detail. Make appropriate abstractions, try to make your classes correspond to sensible entities, but don't get caught up in the question of whether or not something is an object. If it makes sense as a concept, it's probably a reasonable object. Good software engineering is more important than good modeling.

Fundamentally, the difference between OO and procedural programming lies in what entities are most important. In a procedural language, procedures are the important thing, and data is secondary. The basic insight of OOP is that many functions can be expressed as operations on a data type, and that this clarifies the design.

Other benefits spring from this insight. Using polymorphism we can dynamically select semantically similar operations on different data types, and specify data types using inheritance for incremental modification. The essential idea is to place the data type at the center. But not everything fits neatly into this model, and it's not the ultimate answer to all programming problems: it is only an improvement on the preceding model. ■

Alan Knight is a researcher in the Department of Mechanical and Aerospace Engineering at Carleton University, Ottawa, Canada, K1S 5B6. He can be reached at +1 613 788 2600 x5783, or by e-mail as knight@mrco.carleton.ca.

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.

The American Information Exchange Corp. (AMIX), a subsidiary of Autodesk Inc., announced the opening of the first of several key online markets for information and consulting services. At the **AMIX Smalltalk Components and Consulting Market** customers can buy and sell Smalltalk/V, Smalltalk-80, and other object code as well as consulting and training services. AMIX establishes transaction rules, facilitates negotiations, and automates payments and collections.

For more information, contact AMIX, 1881 Landings Drive, Mountain View, CA 94043-0848, 415.903.1000.

Digitalk Inc. has announced a new version of Smalltalk/V for Windows that simplifies the complex task of writing programs for Microsoft's popular Windows environment.

The new version of Smalltalk/V includes support for Windows Multiple Document Interface (MDI), a ToolPane (a row of buttons that perform functions when selected), a StatusPane that displays information on the status of applications, an ObjectFiler for sharing objects with other applications and developers, HelpManager support for non-US character sets, and performance improvements. In addition to standard Smalltalk/V features, the package provides interfaces to Dynamic Data Exchange (DDE), allowing information to be shared between Smalltalk/V programs and other programs, and Dynamic Link Libraries (DLLs), which provide a mechanism for calling code written in other languages from within Smalltalk/V.

For more information, contact Digitalk Inc., 9841 Airport Boulevard, Los Angeles, CA 90045, 310.645.1082, fax 310.645.1306.

Zoom (Zippy Object-Oriented Memory) is a simple object-oriented database written in Smalltalk/V for the 286, Windows, PM, and Mac platforms. Zoom offers variable length keys for random access messages at:, at:put:, removeKey: and seqeuntial messages do:, first, next, prior, and last. A size method is available and class method open: starts any database file while new: guarantees a new file. Zoom works best by providing keyed access to Digitalk Loader/Dumper object representation, but an alternative representation requiring programming is supplied. References between filed objects must be made by name in your application.

For more information, contact Expertek, P.O. Box 611, Clatskanie, OR 97016, 503.325.4586.

HIGHLIGHTS

Excerpts from industry publications

SMALLTALK

. . . If Smalltalk is so powerful, why does it have such a small following compared with C++? Dan Shafer, author of the book *Practical Smalltalk*, suggests that Smalltalk is so completely different from any other development environment that the first reaction of procedural programmers is panic...Smalltalk's classes and methods are not just a class library but an integral part of its environment that makes up Smalltalk. Everything interacts with everything else. This can be quite disconcerting for the beginner, and the fear of breaking something can often serve as the greatest deterrent to learning Smalltalk...Ultimately, we return to the original question: Why Smalltalk? Because you want an environment built around object-oriented programming, not derived from procedural programming. You want an environment that provides extensibility while managing your code. You want the flexibility of an interpretive language in which you can play with and test your code, coupled with the performance of a compiler. You want an interactive debugging environment that lets you inspect and modify your code and variables on the fly with instant results, instead of saving, compiling, and linking between changes.

Why not Smalltalk? William Scott Herndon, UNIX REVIEW, 5/92

PREDICTIONS

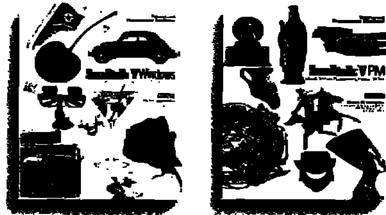
. . . The object-oriented programming revolution may be the beginning of the biggest programming advance in the history of computers. It may prove to be the software equivalent of the microprocessor, allowing the mass creation of more capable, less expensive software. We say "may" simply because it may also be that object-oriented programming is just the beginning of that revolution and will itself be swept away in a comparatively short time by the new technologies it makes possible

Object-oriented methodology, OPEN SOFTWARE JOURNAL, vol.5/no. 1 1992

STRATEGIES

. . . Robert E. Lee said "Plan no more than necessary." His ultimate defeat was probably due more to the implementation of this philosophy than its validity. The problem in development, again, as in war, is how to know when to stop planning and start moving. The answer is never stop planning but never let planning prevent progress. The best methods today facilitate iterative development. Use one with object-oriented techniques for the appropriate tasks to get the most powerful and complete approach available.

Planning, lookahead,and spiraling into control, Adrian Bowles, OBJECT MAGAZINE, 7-8/92



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

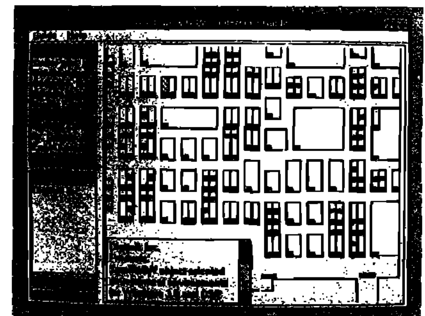
NCR has an integrated test program development environment for digital, analog and mixed mode printed circuit board testing.

MIDLAND BANK

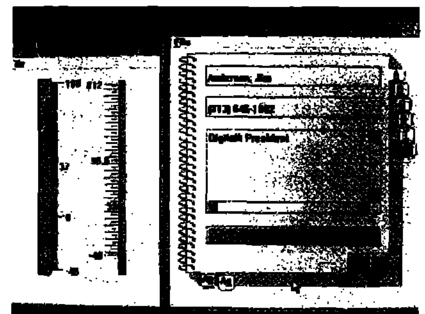
Midland Bank built a Windowed Technical Trading Environment for currency, futures and stock traders using Smalltalk V.

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.