Box 5
folder 17

Smalltalk report, The, 1991-1998

Adele Goldberg papers

Periodicals, technical papers, and articles

X5774.2010, *Box 5* 102739331

1 of 4

The Smalltalk Report

The International Newsletter for Smalltalk Programmers

September 1991 Volume 1 Number 1

THE COMMERCIAL EVOLUTION OF SMALLTALK

By Abdul K. Nabi

Contents:

Over the last decade, Smalltalk has made a dramatic evolution from a visionary software research project into a commercial environment that is spearheading object-oriented programming, the next step in software technology.

The focus of this article is the evolution of commercial Smalltalk from the early market to today's commercial success and what the future may hold. In addition to the evolution of Smalltalk, the changes in the needs and demands of software development that contributed to the spread of Smalltalk in commercial application development will be discussed.

THE EARLY YEARS OF SMALLTALK

Xerox, in the interest of broadening the Smalltalk base, licensed Smalltalk to several hardware vendors (Apple, Hewlett-Packard, and Tektronix) and one startup software vendor (Softsmarts). This groundwork led to the creation of a Smalltalk marketplace.

Xerox sold the Smalltalk environment bundled with its proprietary graphics workstations, which were quite expensive. These workstations pioneered the idea of high-performance, graphical, interactive desktop computers. Xerox used Smalltalk internally to develop custom document and information management applications. One major application created in 1982, was a desktop publishing system for *The New York Times*. The next year, Xerox developed Analyst, one of the first and best known commercial Smalltalk applications.

The Analyst is an integrated information management system incorporating document processing and layout, a spreadsheet, charting, database, hypertext, links, and a world map. The embedded object architecture, level of integration, and seamlessness of the applications is outstanding even by today's standards. Analyst's capabilities are unmatched by any other software package, partly because Analyst is written in Smalltalk. Analyst is still being sold as a commercial, end-user application by Xerox Special Information Systems.

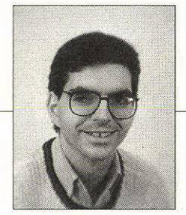
To put the early years of Smalltalk at Xerox in some perspective, Smalltalk-80 was created about the same time the IBM-PC was introduced. The Analyst was introduced two years before the Apple Macintosh.

One of the early users of the exotic Xerox workstation and the Analyst were US intelligence agencies. These agencies required the horsepower, graphics, and high-performance development environment that the Xerox workstation and Smalltalk provided to create interactive analysis workstations. This helped Smalltalk emerge from the lab into the marketplace. However, Xerox did not market these workstations and Smalltalk-80 to the general software development market.

Tektronix was also an early pioneer of commercial Smalltalk, delivering its first Smalltalk in 1985 (which, like Xerox's, ran on a proprietary workstation). Unlike Xerox, Tektronix was actively marketing the environment as a development tool, and at a considerably lower cost (since by 1985 processors like the 68000 that Tektronix used made low-cost workstations possible). Tektronix also used Smalltalk to develop custom software for their workstations (such as a front-end for VLSI test equipment). Although successful

continued on page <None>...

John Pugh



Paul White

EDITORS' CORNER

Welcome to the first issue of *The Smalltalk Report*! The Smalltalk community has long yearned for its own publication. With your help, *The Smalltalk Report* will fill the void.

The use of Smalltalk within industry is expanding rapidly. Only industry insiders know many of the exciting developments that are taking place and, as Abdul Nabi points out in his lead article, they are not permitted to share them. Many companies believe it to be a strategic advantage to be using Smalltalk and don't wish their competitors to be aware they are using it. So, the language that started it all (with apologies to Simula) is now seen by many as the development system of choice for the 1990s. As our good friend Dave Thomas (paraphrasing Winston Churchill) is quoted as saying, "Smalltalk is the worst possible programming environment — until compared with all other programming environments."

Our aim at the *The Smalltalk Report* is to support the growth of Smalltalk as a development vehicle for object-oriented systems and to serve as a focal point for sharing ideas and experiences gained from the employment of Smalltalk technology in areas as diverse as real-time embedded systems and financial systems. By publishing nine times a year, we will be able to bring you timely information on new software releases of all dialects of Smalltalk, third-party products, class libraries, books, industry news, etc. We will address all aspects of application development with Smalltalk, e.g., project management, analysis and design, development tools, language issues, metrics, performance issues, and education and training.

In the lead article of our premiere issue, Abdul Nabi takes us on a tour through time — from the early days of Smalltalk at Xerox PARC to current implementations and applications. He discusses the issues faced by Smalltalkers in the past, problems that have been solved, and the challenges that lie ahead. He explains how, and why, the commercial evolution of Smalltalk has unfolded in the manner it has and speculates where this evolution will lead.

In this first issue, we also introduce two of our regularly appearing columns. In "Getting Real," Juanita Ewing discusses the issues of developing industrial strength applications with Smalltalk. In her first column, she addresses the pros and cons of employing class ownership as a vehicle for code management within a programming team. In their GUI column, Greg Hendley and Eric Smith tackle the topic of graphical user interfaces. In the first installment of a two-part column, they discuss giving application windows dialog box functionality in Smalltalk/V PM. In future issues, watch out for other columns; on design from Rebecca Wirfs-Brock and on "Smalltalk with Style" from Ed Klimas and Suzanne Skublics.

Our authors and columnists are willing to stand up and be counted, expressing their own personal, sometimes controversial, opinions. To make this forum truly effective, we encourage you to "jump into the foray" and let your ideas be heard. Use our "Messages" corner as a soapbox to vent your own opinions. In this issue, Allen Wirfs-Brock laments the absence of a conference where Smalltalk users and developers can get together and share their work.

Also in this issue: Charles Rovira suggests enhancements to the compress changes facility in Smalltalk/V Windows, Ralph Johnson describes the Typed Smalltalk project at the University of Illinois, and Jim Salmons reviews the WindowBuilder/V product from Acumen.

The Smalltalk Report is written by Smalltalkers for Smalltalkers; we encourage you to contribute.

Enjoy the first issue!

John Pugh and Paul White
Editors

The Smalltalk Report (ISSN# 1056-7976) is published 9 times a year, every month except for the Mar/Apr, July/Aug, and Nov/Dec combined issues. Published by COOT, Inc., a member of the SIGS Publications Group, 588 Broadway, New York, NY 10012 (212)274-0640.

Mailed First Class. Subscription rates 1 year, (9 issues) domestic, $65, Foreign and Canada, $90, Single copy price, $8.00. POSTMASTER: Send address changes and subscription orders to: The Smalltalk Report, Subscriber Services, Dept. SML, P.O. Box 3000, Denville, NJ 07834. Submit articles to the Editors at **91 Second Avenue, Ottawa, Ontario K1S 2H4, Canada.**

ParcPlace announces 4GL application development tool for Objectworks\Smalltalk

ParcPlace systems announced the availability of FACETS, a development tool for use when building applications in Objectworks\Smalltalk Release 4. FACETS, supplied by Reusable Solutions, was designed to help create screen-based, data-intensive applications such as order entry, financial processing, and other database-oriented 4GL applications. In conjunction with Objectworks\Smalltalk Release 4, FACETS provides an extendable object-oriented 4GL development environment and series of on-screen forms that guide the user through the rapid generation of interface components.

FACETS is fully compatible with Objectworks\Smalltalk Release 4 and fully portable across all supported platforms, and allows full connection to Servio's Gemstone interface for powerful database connectivity.

For further information, contact ParcPlace Systems, 1550 Plymouth St., Mountain View, CA 94043, (800) 759-PARC.

Object Technology International announces an object-oriented team development environment for OS/2 and Windows

Object Technology International, Inc. (OTI) announced the immediate availability on OS/2 and Windows of Release 1.0 of ENVY/Developer, an object-oriented team programming environment. With ENVY/Developer, development teams using Smalltalk may work concurrently on both OS/2 and Windows, sharing code and data using the tools provided by the environment.

The environment supports the full manufacturing lifecycle including prototyping, development, interactive debugging, performance analysis, packaging/delivery, and maintenance of large systems written in Smalltalk, and provides all tools required to realize the benefits of object-oriented software development. ENVY/Developer is currently the only toolset for delivering large systems incorporating advanced object-oriented technology.

For further information, contact Object Technology International, Inc., 1785 Woodward Dr., Ottawa, Ontario K2C 0P9, Canada, (613) 228-3535, or fax (613) 228-3532.

PUBLISHER'S NOTE

"Isn't it time for an independent publication devoted exclusively to Smalltalk users" is a question I'm frequently asked at conferences. Even though Smalltalk is celebrating its tenth anniversary this fall (since *Byte*'s landmark issue lauding the language) there's been a surprising paucity of editorial coverage devoted to Smalltalk in any publication since.

We at SIGS Publications have seen a recent resurgence in the interest in and usage of Smalltalk. It remains the archetype of a pure and fully integrated O-O development environment as OOP explodes in the 1990s. According to Ovum's *Object Technology Sourcebook*, Smalltalk sales (in the US and Europe) are currently $21 million and are expected to double to $40 million by 1993 — making it one of the fastest-growing languages.

The time has come for an independent forum devoted exclusively to Smalltalk users' informational needs. *The Smalltalk Report* will publish over 200 pages of need-to-know information on Smalltalk during its first volume year. Our editorial mission, simply stated, is to stimulate, track, and evaluate Smalltalk usage on a worldwide basis.

Welcome to the premiere issue. It represents hundreds of hours of thinking, writing, and research. By reading *The Smalltalk Report*, you can quickly benefit by gaining access to nowhere-else-found techniques, advice, ideas, source code, and "insider news" — a veritable goldmine of consolidated information. You can rely on what you read in *The Smalltalk Report* to be timely and accurate. We publish it with the same editorial integrity as we do our sibling publications, the *Journal of Object-Oriented Programming*, *Object Magazine*, *The C++ Report*, the *Hotline on Object-Oriented Technology*, and *The International OOP Directory*.

I encourage you to contact us regarding your opinion of this issue and what you'd like to see in upcoming issues. Your feedback is valuable to us as the newsletter evolves.

I invite you to plug into the insiders network of Smalltalk developers by becoming a Charter Subscriber. Join our family of well-informed readers. We look forward to serving your informational needs. Enjoy the premiere issue!

Sincerely,

Richard Friedman

Richard P. Friedman
Group Publisher

continued from page 1...

in-house, Tektronix, primarily a test equipment manufacturer, had difficulty marketing their Smalltalk workstation. By 1985, developers and organizations were standardizing on mainstream personal computers and workstations; thus, the appeal of a custom workstation was limited. However, Tektronix developed a large group of people experienced in developing and marketing Smalltalk and object-oriented development that would later seed several successful companies in the Smalltalk and OOP market.

Digitalk introduced Methods, a text-based Smalltalk, in 1985, and then Smalltalk/V, a graphics version, in 1986. The most significant feature was that Digitalk's Smalltalk ran on the popular IBM-PC. By providing a low-cost Smalltalk for the IBM-PC, Digitalk expanded the base of Smalltalk users, many of whom were building prototypes or custom applications. The success of these early developers spread the use of Smalltalk as a commercial development environment.

In early 1986, a company by the name of Softsmarts brought the first Smalltalk-80 for the IBM PC/AT to the market. Like Digitalk, the Softsmarts version proved that Smalltalk could run on low-cost personal computers. Softsmarts was also the first Smalltalk-80 to incorporate color graphics and external language support. However, the PC marketplace for Smalltalk became dominated by Digitalk with lower-cost versions of their Smalltalk/V product (which included the same feature set and ran on eight-bit PCs).

About the same time, the group within Xerox PARC that had created Smalltalk wanted to spin off a company that would focus on marketing Smalltalk. From that drive, ParcPlace Systems was born. The first few years were spent creating the infrastructure of the company and creating portable commercial versions of Smalltalk for PCs, Macintoshes, and UNIX workstations.

Although the early and mid-1980s laid the groundwork for the future growth of Smalltalk, both the state of the computer industry and Smalltalk itself prevented the widespread acceptance and use of the language.

One perception of Smalltalk that remains to this day is that it performs poorly compared to standard languages. Much of the perception is based on the fact that early versions of Smalltalk were interpreted and included automatic storage reclamation (garbage collection). What is interesting to note is that even early versions of Smalltalk performed quite well (most people made performance statements without direct experience). Much of this performance is based on the fact that Smalltalk is best suited for complex, interactive information analysis and management applications. In simpler applications, the overhead of Smalltalk becomes a factor that makes it uneconomical for those applications. As the application becomes more interactive or complex, the power of Smalltalk becomes a key benefit in both development time and cost. Also, performance can be improved since the lower-complexity code that is created by using Smalltalk can be optimized

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.

Logic Arts announces VOSS: virtual object storage system for Smalltalk/V

Logic Arts' virtual object storage system, VOSS, is available now for Smalltalk/V 286. Voss object management facilities include: persistent storage, transparent access, virtual collection and virtual dictionary, multikey access, a class restructure editor, and import/export, in which administration facilities provide for backup, restore, renaming, import/export between machines, or access over a network. VOSS also features performance tuning: the control panel allows cache size and other parameters to be tuned for optimum performance, according to the degree of object volatility and random v. sequential access to virtual collections. Many of the new classes are independently reusable.Smalltalk/V286 source code is supplied. VOSS requires Smalltalk/V286 and MS-DOS.

For further information, contact Logic Arts, Ltd., 75 Hemingford Rd., Cambridge CB1 3BY, UK, (0223) 212392, or fax (0223) 245171.

Tigre ships multiplatform rapid GUI application builder

Tigre Object Systems, Inc. of Santa Cruz, CA, is now shipping the Tigre Programming Environment. Tigre implements the capability to build graphical user interface applications quickly for instant use on multiple computer platforms and heterogeneous networks. Color applications created by Tigre run without modification on Windows 3.0. Macintosh II, Sun/3, Sun SPARCstation, IBM RS/6000, Digital DECstation, Hewlett-Packard's HP 9000 Series 300 & 400, Apollo Series 2500, 3500, 4500, Sequent superminis, and on mixed networks of these. Additional platforms will follow. Tigre, a fully object-oriented system, uses Objectworks\Smalltalk Release 4 by ParcPlace Systems as its scripting language.

For further information, contact Tigre Object Systems, 3004 Mission St., Santa Cruz, CA 95060, (408) 427-4900, or fax (408) 457-1015.

Digitalk announces Smalltalk/V developer conference

Digitalk, Inc. announced their first developers' conference, Smalltalk/V Dev Con '91. The conference will take place September 11–13 at the Universal City Hilton and Towers in Universal City (Los Angeles), CA. Sponsored by Digitalk and BYTE magazine, the conference will include a wide range of technical topics, panel discussions, speakers, and product demonstrations. Events include: sessions on design, management issues, application delivery, Smalltalk/V internals, integrating with other languages, integrating with other products, etc., as well as panel discussions, and industry guest speakers.

For further information, contact Digitalk, Inc., 9841 Airport Blvd., Los Angeles, CA 90045, (213) 645-1082, or fax (213) 645-1306.

Instantiations announces new engineering tools and version management for Smalltalk

Instantiations, Inc. announced that it has developed a powerful new set of software engineering tools to support developers using Objectworks\Smalltalk called Application Organizer Plus. The product is an integrated set of tools that give Smalltalk users new ways to structure applications, manage code, and optimize reuse and was specifically designed to provide these capabilities without sacrificing the freedom and high level of interactivity that are the essence of Smalltalk programming.

Application Organizer Plus provides the Objectworks\Smalltalk developer with version management, improved code modularity, enhanced reusability, smaller delivered applications, new browsers, and workspace enhancements.

For further information, contact Instantiations, Inc., 921 S.W. Washington, Ste. 312, Portland, OR 97205, (503) 242-0725.

Digitalk ships new release of Smalltalk/V Windows

Digitalk, Inc. announced a new release of Smalltalk/V Windows, which combines Digitalk's widely used object-oriented programming environment with Microsoft Windows 3.0. Smalltalk/V Windows Release 1.1 contains an icon editor, performance improvements, better memory utilization, and many new programming examples demonstrating usage of Windows features.

Smalltalk/V Windows includes standard Smalltalk/V features such as source code browsers, inspectors, and push-button debuggers. In addition, Smalltalk/V Windows provides interfaces to dynamic data exchange (DDE), allowing information to be shared between Smalltalk/V programs and other programs and dynamic link libraries (DLLs), providing a mechanism for calling code written in other languages from within Smalltalk/V. Smalltalk/V Windows source code is compatible with Digitalk's Smalltalk/V PM programming environment for OS/2.

For further information, contact Digitalk, Inc., 9841 Airport Blvd., Los Angeles, CA 90045, (213) 645-1082, or fax (213) 645-1306.

Digitalk announces royalty-free runtime

Digitalk, Inc. announced new versions of Smalltalk/V DOS and Smalltalk/V Mac that include royalty-free runtime. Smalltalk/V Windows and Smalltalk/V PM are already royalty-free.

The Smalltalk/V DOS Version 3.0 runtime system allows developers to create standalone executable applications and includes integrated EGA/VGA color. Registered users of earlier versions of Smalltalk/V may purchase an upgrade that includes a new manual.

The new version of Smalltalk/V Mac allows developers to create standalone, double-clickable applications with no additional royalty payments. Prior to this new policy, there was a per-copy charge for runtime applications. Registered users of earlier versions of Smalltalk/V may purchase an upgrade.

For further information, contact: Digitalk, Inc., 9841 Airport Blvd., Los Angeles, CA 90045, (213) 645-1082, or fax (213) 645-1306.

WHAT THEY'RE SAYING ABOUT SMALLTALK

Excerpts from industry publications

... Smalltalk/V has been realized in DOS, Macintosh, and Windows versions. Much of the code can be used across all the environments. Objects can be stored in text form, and "filed out" and "filed in" from system to system. Because Smalltalk is an interpreter, these objects can be introduced into a running system. The potential exists to network together Smalltalk systems on a number of different platforms, and to let them exchange objects in real time. This is not something you can do with C++. Smalltalk/V has the most thorough tutorial of any of the packages reviewed here. The manual is a complete course in the language, and the example files give you working code for most applications. Smalltalk is not like other computer languages. Instead of being like a musical score, it is more like a jam session in which you create both new instruments and new musicians as you go along. When you've constructed your band, you're ready to play. Smalltalk is extraordinarily interactive, and the ideal environment for creative people. Accessibility is excellent, limited only by Smalltalk's unusual syntax. Every element of Smalltalk, from object creation to debugging and running the application, takes place with a single Windows application. Integration is total. The assistance to clear thinking, Smalltalk's clean handling of the Windows environment, its integration and rich data taxonomy, and its potential for interplatform development, make Smalltalk/v the winner among all the packages we have surveyed ...

Breaking into Windows, Birrell Walsh,
Microtimes, 6/19/91

... Only the interpreted object-oriented systems such as Smalltalk, object-oriented Lisp, and various proprietary object-oriented development systems, have a clear edge over Nextstep in programmer productivity. Because these systems are unsuitable for producing commercial applications due to their poor performance, huge size, or restrictive licensing policies, it is hard to refute the commonly heard claim that the Nextstep environment is the most productive mainstream development environment available today ...

The Next Next, Scott Raney,
UNIX World, 7/91

... Smalltalk is not about to replace COBOL, but it is finally maturing into a viable choice in application development, especially for users looking for a tool to speed development of advanced graphical user interfaces in client/server applications ... As a dynamically compiled language built on reusable objects and a virtual interface that uses machine-independent, intermediate code, Smalltalk is also easily portable between the platforms it supports ...

... there is still no widely accepted development methodology for Smalltalk or for any other object-oriented environment. In addition, many users are still making the transition to the relational model and structured programming techniques. "Most [IS developers] still don't know what to do with objects. They're still traumatized from making the migration to the RDBMS," says Natasha Krol, application program director at the Meta Group in Stamford, Conn. Smalltalk also faces increasingly stiff competition not only from other object-oriented languages such as C++ but also from new GUI-building tools, such as Easel from Easel Corp, and Actor from Whitewater Group ...

Smalltalk Grows Up, Jeff Moad,
Datamation, 7/15/91

IBM will postpone its scheduled announcement of support for object-oriented technology in AD/Cycle until later this year ... IBM had planned to announce by the end of this month support for the Digitalk Smalltalk language in its strategic software development environment. When it does make the announcement, IBM said, it will also provide a more substantial statement of direction for AD/Cycle and object-oriented languages as well as methodologies ... The AD/Cycle announcement will focus more on how object-oriented technology will affect the whole development life cycle ... rather than on an individual product ... IBM still plans to include Smalltalk in AD/Cycle and will also recognize C++ as an AD/Cycle language ...

IBM puts off object-oriented support, Rosemary Hamilton,
Computerworld, 6/17/91

... GUIs, however, are not a prerequisite of OOP programs, But, the two have become closely identified because GUIs increase the size and complexity of programs to the point where traditional programming methods cannot manage them effectively. OOP, on the other hand, can easily accommodate the programming of GUIs. In fact, Smalltalk, one of the first completely object-oriented languages, incorporates a graphical environment of menus, windows and scroll bars ...

Programming with Modules,
Chemical Engineering, 6/91

more easily than the large, complex amount of code created in traditional languages.

Another major obstacle to the widespread use of Smalltalk was a lack of acceptance of both graphical user interfaces and the object-oriented paradigm. Most computing was being done on either terminals or text-based PCs. Graphics were reserved for video games and exotic applications. Much of this was due to the fact that high-performance hardware and high-resolution graphics displays were not commonly available. PCs were being used in data entry, or simple analysis, and not to provide highly interactive interfaces or to solve complex problems. Thus, the range of applications that Smalltalk is ideally suited for were not being widely developed.

Many early projects done in Smalltalk were either prototypes or systems that evolved through many iterations. O-O analysis and design methodologies along with good implementation strategies were still forming (many of these early projects contributed to this process). However, in comparison to traditional development languages, Smalltalk appeared cavalier, undisciplined, and immature. The speed perception and lack of object-oriented analysis and design methodologies created the perception that Smalltalk, although good for rapid prototyping, could not be used in a disciplined way to create robust, high-quality, commercial applications.

During the middle to late 1980s, desktop computing and GUI-based interfaces became accepted. Smalltalk was behind in integrating with the standard GUIs that were emerging, continuing to provide its own nonstandard GUI. Also, Smalltalk was a closed language, not allowing interfaces to other languages or libraries. However, now all versions of Smalltalk from both ParcPlace and Digitalk integrate with the standard windowing systems and external languages. As a result, Smalltalk provides one of the best environments for development of host-based applications (given the complexity of GUI programming interfaces).

Another major block was the lack of Smalltalk developers, tools, training, and support services. These were areas of the market that had to grow to make Smalltalk a viable commercial development environment.

SMALLTALK TODAY

Over the last five years, the Smalltalk industry has grown and most of the hurdles have been cleared. Smalltalk has become more widely used in commercial software development. This is due not only to changes in the Smalltalk environment itself, but also to the software development marketplace as a whole.

Changes in the overall marketplace have played a key role in the success of Smalltalk. Time-to-market, adaptability, and cost control have become increasingly crucial factors in overall business success. It is this business environment that has accelerated the acceptance of object-oriented technologies and Smalltalk. Information systems managers today are more interested in solutions than in the technology employed. Many clients we work with that would have never considered Smalltalk as a development and delivery language a few years ago are now pursuing aggressive Smalltalk strategies.

The personal, highly interactive, graphical environment that Smalltalk pioneered is now accepted and several GUIs are widely in use. One result of the acceptance of GUIs is a further increase in software complexity and development costs. Developers now have to deal with the large library of APIs that GUI and operating system have. Actually, the Smalltalk windowing system, which has been considered difficult to use, looks absurdly simple when compared to standard windowing system libraries.

> **"Many clients ... that would have never considered Smalltalk as a development and delivery language a few years ago are now pursuing aggressive Smalltalk strategies."**

Both ParcPlace and Digitalk have introduced versions of Smalltalk that access and use the host GUI and APIs. When there were no standard windowing systems, Smalltalk provided its own. This, of course, was unacceptable once standards for windowing systems were established. Both Digitalk and ParcPlace versions of Smalltalk run using the host windowing system and allow access to the host API and external language functions (i.e.,Windows and Presentation Manager DLLs).

One of the main differences between Digitalk and ParcPlace's versions of Smalltalk is how they provide host integration. Digitalk's Smalltalk V Mac, V Windows, and V PM provide tight integration with their host and use the host environment controls and libraries (windows, menus, buttons, and so on). ParcPlace's Objectworks/Smalltalk Release 4 uses only the higher-level and portable host services (windows, fonts, and graphics) to provide image portability across platforms and operating systems. Objectworks/Smalltalk does allow the developer to access the host's controls and libraries at the expense of portability.

The amount of computing horsepower that sits on the average desktop today exceeds the horsepower that came with the original $100,000 Smalltalk machines from Xerox. This allows the user to use the system in an interactive graphical environment to solve increasingly complex problems. However, the cost and development time of software using traditional methods has not kept pace, leaving idle MIPS on the desktop. Smalltalk allows an effective, efficient, and cost-effective way to develop interactive applications that solve complex problems.

Both Digitalk and ParcPlace have also improved performance of the system by switching from a purely interpreted environment to executing compiled code. The performance of garbage collection has also been increased. In a variety of applications, particularly highly interactive and complex analysis, Smalltalk actually performs as well as or better than systems developed with traditional languages.

An example of high-performance Smalltalk in a commercial application is HPMS, a system developed for Hewlett-Packard by Knowledge Systems Corporation. The HPMS system is a complex process modeling tool primarily designed for manufacturing. It includes heavy computation and graphics for flow autorouting and diagramming. Most who see the system believe that it actually was written partly or entirely in C. However, HPMS is implemented entirely in Smalltalk without the use of C or assembly code. More information on the HPMS system can be found in Robert Whitefield and Ken Auers' article "You can't do that with Smalltalk! Or can you?" in the May/June 1991 premiere issue of *Object Magazine*.

Besides the Smalltalk language vendors, several other companies have formed to provide tools, training, consulting, and support for Smalltalk. Without this framework of companies providing the supporting products and services, corporations could not make the commitment and investment in Smalltalk.

One company, Object Technology International (OTI), provides team development and source code control tools for Smalltalk (essential for large-scale commercial development). OTI has also used Smalltalk successfully in ROM-based embedded controller applications, where typically low-level languages are used.

TODAY'S OBSTACLES FOR SMALLTALK

Many of the companies that are using Smalltalk are very secretive about their use (to the point of not allowing any Smalltalk books to be visible in offices). These companies view the use of Smalltalk as a strategic competitive advantage. Unfortunately for those in the Smalltalk industry, this reluctance to share success stories makes it difficult to promote wider use of the language through examples. Often people in the Smalltalk industry, when talking about Smalltalk's success, must be vague with lines like, "All sorts of companies are having tremendous success with Smalltalk, but we can't tell you about any of them."

As larger projects are being developed with Smalltalk (by companies we can't talk about), more time is being spent on analysis, design, and software quality. When used for prototypes, analysis and design are not significant issues. Still, for high-quality production software, Smalltalk requires design, testing, and iteration. Even today, many users first developing with Smalltalk get enamored of the enormous productivity gains of Smalltalk and try to turn functional prototypes into commercial software (which ends up being low in quality, difficult to maintain, or taking longer than expected). The process of managing the Smalltalk software lifecycle and then reuse of code are still issues. As more experience in managing the high productivity of Smalltalk is compiled, issues such as reuse and quality will be better understood.

Companies now making the investment in Smalltalk development face the difficulty of finding resources and education. The number of experienced Smalltalk programmers is limited, and competition for those developers is heavy. In addition, training in-house developers in object-oriented technology and Smalltalk takes approximately two months. After the initial training period, six months of use is required before enough experience is developed to create quality commercial software. Managers have difficulty accepting these time frames, given the pressure to deliver. Often this pressure is the reason for using Smalltalk.

SMALLTALK TOMORROW

Over the next few years, several significant products will come out using Smalltalk. Smalltalk development and product success stories will be published (many in The Smalltalk Report). The base of users and projects will expand both in organizations already using Smalltalk and in new ones.

The Smalltalk industry will expand with more companies being formed to provide products and services, particularly developer training, analysis and design tools, code generation, application frameworks, and tools to manage large-scale reuse of code.

Several companies will deliver integrated analysis, design, development, and lifecycle management tools developed in Smalltalk. These tools will push object-oriented application development into a more disciplined and efficient level, particularly in large organizations.

For application developers in both large and small organizations, more development tools will be delivered. Interface builders and application frameworks such as Acumen's Widgets and Tigre Object Systems' Tigre Programming Environment are already in use building successful commercial applications.

In the software development community, there is always the tendency to find the best technology. Currently, in the object-oriented arena, many are looking for a winner, be it C++, Smalltalk, Eiffel, and so on. The history of software shows us that there isn't a winner, just as there isn't any best automobile. There will be a variety of languages and tools to support various types of development. Smalltalk will find success in commercial applications, particularly in interactive desktop analysis applications, where the power of Smalltalk is best applied. ❁

REFERENCE

[1] Whitefield, R. and K. Auer. You can't do that with Smalltalk! Or can you? *Object Magazine*, 1(1), 64–69, 1991.

Abdul K. Nabi can be reached at Knowledge Systems Corporation, 114 MacKenan Dr., Ste. 100, Cary, NC 27511.

Experienced Smalltalk/V Windows and Smalltalk/V PM developers probably noticed that WindowBuilder uses WBTopPane as the parent of application windows rather than the more flexible and powerful ViewManager class. In many circumstances, the multi-window views supported by ViewManager designs are not required. When use of ViewManager is desirable, it is possible to add each WBTopManager subclass as a view of your application's ViewManager instance and set the owners of all Subpanes of your WindowBuilder windows within your window's TopPane to the ViewManager instance. While this is possible, I would like to see a clean and easy "Link to ViewManager Instance" option in a future release of WindowBuilder.

Also, there is no easy way to save WindowBuilder designs as subclasses of other WindowBuilder subclasses. I would like to be able to encapsulate reusable instance variables and methods for a DDE client WindowBuilder window in a new abstract subclass of WBTopPane. New DDE-based WindowBuilder designs could be created as a subclass of this abstract class. Currently, the only way to do this is to create your new design as a subclass of WBTopPane, file it out, remove it, edit the source and file it back in as the subclass of your abstract subclass of WBTopPane.

Where truly high performance is required or where multiple instances of a window or dialog may be active at one time, it is often desirable to compile a window or dialog using the Microsoft Resource Compiler from the Windows Software Development Kit or similar tool. Stored in a dynamic link library (DLL), such resources blast onto the screen when created and may take advantage of DLL shared run-time functionality. A "Write WindowBuilder Design to DLG Script" which could be fed to the resource compiler would be useful.
Finally, WindowBuilder does not fully implement the user interface standards of the Windows and Presentation Manager supported Common User Access (CUA) protocol. CUA defines the "proper" way a keyboard interface should work in terms of tabs between control groups and arrow keys moving within a group's items, etc. While a WindowBuilder window may have the "look" of a CUA-compliant window or dialog, the user access interaction misses the mark in terms of these subtle "feel" requirements.

HOW WINDOWBUILDER STACKS UP

WindowBuilder is a welcome addition to any Smalltalk/V Windows developer's toolkit. WindowBuilder will enhance the productivity of the new as well as experienced Smalltalk/V Windows developer.

By comparison, Digitalk's forthcoming Smart Parts product (demoed for nearly a year as the "Look and Feel Kit") has the potential to establish an entirely new programming paradigm for Smalltalk application development. Smart Parts will be a radical departure from traditional Smalltalk development procedures. While Smart Parts will be revolutionary, WindowBuilder is a solid evolutionary extension to Smalltalk/V development.

Try it. You will like it. Thanks, Acumen, and keep up the good work. ❁

PRODUCT INFORMATION

WINDOWBUILDER
RETAIL PRICE: $149.95
SYSTEM REQUIREMENTS:
SMALLTALK/V WINDOWS,
MICROSOFT WINDOWS 3.0 OR LATER

ACUMEN SOFTWARE
2140 SHATTUCK AVENUE, SUITE 1008
BERKELEY, CA 94704
(415)-649-0601

Jim Salmons is President of JFS Consulting of Lexington, South Carolina. JFS Consulting specializes in the documentation of object technology products and object-based user interface revision control systems. With his partner, Timlynn Babitsky, Jim is coeditor of The International OOP Directory, *published by SIGS Publications. Jim and Timlynn are also Exhibits Cochairs of the annual ACM OOPSLA Conference.*

GETTING REAL

Juanita Ewing

Should classes have owners?

As Smalltalk engineering projects grow larger, the need for reusable code increases. Developers need to build larger applications even faster. The easiest way to increase the capabilities and scope of an application is to reuse more classes. Large applications require teams of Smalltalk programmers to glue these reusable classes together and write some application-specific code, too.

WHAT IS A REUSABLE CLASS?

Classes are reusable in two ways: as a client making instances or as the basis for new subclasses. The characteristics of these two kinds of reusable classes are different. For client use, you want a fleshed-out and general class. For subclassing, you want a minimal and flexible class. Beyond these characteristics, how do you tell if a class is reusable? To paraphrase Ralph Johnson, a class isn't reusable until proven reusable. That means it has been used in more than one application.

It takes extra time and effort to write classes that are reusable. This extra effort is a separate programming activity. Developers caught up in deadlines for delivering an application often don't have the time necessary to flesh out and polish their classes. For example, developers will initially create a single class that should be refactored into a combination of an abstract class and a concrete class. The concrete class can be reused by making instances of it and the abstract class can be reused by making new subclasses derived from it. The reusability of classes written with the goal of multiple uses is much greater than those written for specific roles in an application.

In conjunction with supporting teams of developers, some Smalltalk environments actively promote the creation of reusable classes. One of the goals of these environments is to separate application engineering from the creation of reusable units of code.

IS CLASS OWNERSHIP A GOOD BASIS FOR PROMOTING THE CREATION OF REUSABLE CLASSES?

Suppose each class is owned by a single developer. The theory is that an owner feels responsible for and will take the extra effort to make a class truly reusable. The creation of reusable classes is important to the entire organization as well as the developers. Programming environment capability by itself is not enough. To back up this capability in the programming environment, the developers' organization must reward the production of reusable code. Responsibility and ownership are established management techniques for motivating employees. It's become common practice in manufacturing environments to give employees more responsibility and have them provide input about the manufacturing process. Employees don't just screw on lug nuts anymore.

Let's assume the owner of a class is rewarded for producing a reusable class. What if another developer finds a bug in that class, or thinks of a useful extension? In a system with class ownership, the owner writes the code to fix the bug or writes a new method. He is the one who is motivated to make the class more reusable.

WHO IS BEST QUALIFIED TO FIX THE BUG OR WRITE THE NEW METHOD?

In the case of the bug, the best qualified person may be the developer who detected the symptom of a problem and isolated the error. After the detective work, fixing the bug may be simple. And, sometimes it is difficult to reproduce a bug. In the case of the new method, maybe the person who thought of the extension knows best how to implement it. Maybe in both cases the owner and the person suggesting the change need to work together to come up with the best solution. The best qualified person depends on the situation. Flexibility in the programming environment is critical.

Systems with class ownership are not flexible. Even the motivational aspects are wrong for flexibility. What is the motivation for developers who are not owners?

DO CLASSES EXIST IN ISOLATION?

When a class is part of an application, it interacts, or collaborates, with other classes. Sometimes the collaboration is part of a framework. For example, a view and a controller collaborate as part of the MVC framework. An instance of view is never used alone. It is always paired with a controller. Because of the relationship between these two classes, coupled with the fact that modifications in one class will probably require corresponding modifications in the other class, there is a strong reason for the same developer to own both of these classes. It makes sense that any related classes should also be owned by the same developer. Evidently all parts of a framework should be owned by the same developer.

Continuing this example, what about the view's relationship with its model? Some views have a close connection with their

models. This argues that the model should be owned by the same developer that the view and controller are owned by. And yet in different applications the same view may collaborate with different models. Are all of those models owned by the developer that owns the view? Class ownership doesn't take into account the flexibility required by multiple applications.

A subclass is closely related to its superclass. If the behavior of a class changes, there may be ramifications in the subclass, requiring corresponding changes in subclasses. This implies that the same developer should own classes that are hierarchically related. Obviously, if one developer owns the entire image, we aren't talking about teams of Smalltalk programmers anymore.

If classes have owners and related classes are owned by the same developer to improve the efficiency of the team, how do you devise a reasonable partitioning if the ownership is restricted to a single developer per class? The answer is, you can't. The goal of grouping related classes conflicts with the goal of distributing classes to individual owners.

> The advantage of multiple developers is to allow multiple perspectives and therefore create more general classes.

DO MULTIPLE DEVELOPERS AFFECT THE QUALITY OF CLASSES?

Close collaboration between developers is important in the production of reusable classes. People who are working together tend to be more creative. Multiple perspectives increase the likelihood of more general abstractions. Multiple developers are an advantage. The result of multiple developers is classes that are well fleshed out and suitable for client use and classes that are general abstractions suitable for subclassing.

SHOULD CLASSES BE ACCESSIBLE TO MULTIPLE DEVELOPERS?

The programming environment needs to promote developers working together. One way to do this is to make classes accessible to multiple developers. That way, each developer could make changes when most appropriate. If you have one owner, what do you do when that owner goes on vacation? What if the owner is ill at a critical time in the project? The programming environment should make it easy to implement contingency plans to keep a project going.

Since a reusable class is produced by a team of people, the entire team should be rewarded. Team programming environments usually have author designations for accountability. Outstanding efforts will continue to be noticed in these environments because of accountability features.

HOW DOES THE PROGRAMMING ENVIRONMENT KEEP THINGS FROM FALLING BETWEEN THE CRACKS?

How do you ensure that the entire class hangs together? You don't want to end up with classes that are a hodgepodge of functionality. Some automatic checks could be installed to produce warnings if, e.g., a method contains no references to self or instance variables.

Most of the consistency checks for a class cannot be automated at this time. A human still needs to browse and understand a class to see if it follows basic design principles. In a cooperative team, this responsibility can be shared. Peer reviews, or more formal code reviews, are an essential part of team efforts.

The programming environment should be able to restrict the set of developers for a class to avoid unauthorized modifications. Many operating systems offer these kinds of limitations. A team programming environment could be even more selective. Also, it is a good idea to place at least one experienced person with a group of inexperienced people. People who have good rapport generally program together well.

A programming environment for teams of Smalltalk developers should promote the creation of reusable classes by rewarding all developers. The advantage of multiple developers is to allow multiple perspectives and therefore create more general classes. Another benefit of multiple developers is more apparent in the final stage of the software lifecycle. Classes that are developed by multiple programmers are therefore understood by multiple programmers. It is easier for the organization to maintain classes because more than one person has the knowledge and understanding required for the job. ❁

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

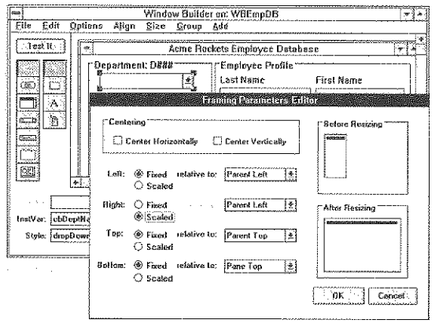


Figure 2. WindowBuilder's Framing Parameters Editor.

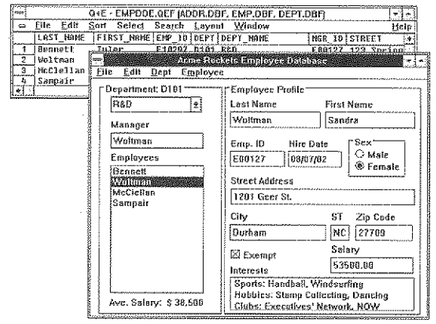


Figure 3. A WindowBuilder-built DDE client application window and its Q&E database server.

Pane subclass at any time and a **Test It** button is provided to generate an instance of your design.

Once you have the design worked out, you may then open a Class Browser on your window's WBTopPane subclass. To complete the implementation, you simply complete the "shell methods" which WindowBuilder generates based on your **when:perform:** and menu item action specifications.

To make all these WindowBuilder features immediately accessible to you, a cogent manual is provided. It includes an overview of the components and functionality of graphical user interfaces, a "Quick Peek" introductory tutorial, a user's guide, an extended example tutorial, a reference section and an index. WindowBuilder is so intuitive, however, that you hardly need the documentation.

USING WINDOWBUILDER TO CREATE A DDE DATABASE CLIENT APPLICATION

About three-quarters of my development session was spent implementing the DDE communication between Smalltalk/V and Pioneer Software's Q&E database engine (Fig. 3). The development of the window design was truly painless using WindowBuilder. Since WindowBuilder generates empty methods based on the control and menu event specifications of your design, it is essentially a "fill in the gaps" process to make the application fully functional.

Had I not been using WindowBuilder, I anticipate my experimental development effort would have easily doubled. WindowBuilder makes Smalltalk/V Windows a viable choice as a consultant's rapid application development environment.

WINDOWBUILDER'S BRIGHT SIDE

WindowBuilder is a vastly improved way to develop a Smalltalk/V Windows user interface when compared to writing raw source code. As a consultant, I would only recommend Smalltalk/V Windows for corporate client development projects if it were enhanced with WindowBuilder.

The Framing Parameters Editor and the Align menu features of WindowBuilder are particularly useful and are often not implemented as well in other user interface builders which I have used.

WindowBuilder is extensible. WindowBuilder is provided in source code and its interface includes a facility for adding your own new Subpane classes. If you create, or purchase, a set of interface components such as ToggleSwitch or ThermometerGuage objects, you could include them in your WindowBuilder designs.

Acumen supplies a WindowBuilder run-time file. Once you have an application built based on a WindowBuilder user interface, you can create a lean image with the classes and method changes required to implement the interface but not the WindowBuilder tool itself.

> **WindowBuilder is a vastly improved way to develop a Smalltalk/V Windows interface when compared to writing raw source code.**

A WINDOWBUILDER WISH LIST

The most glaring problem I had with Version 1.0 was the lack of a Z-order editor. Windows uses a Z-order list to determine the order through which the window "focus" will progress under keyboard control. In a data entry application, you often want to make an entry and tab to the next logical field. It is surprising that Acumen did not provide any means to control and reorder this all-important aspect of a window or dialog design. The current workaround for the lack of a Z-order editor is to cut and paste the **addSubpane:** blocks in the **addSubpanesTo:** method. In a window as complex as the DDE Database example, this is incredibly tedious.

reviewed by Jim Salmons

WindowBuilder: An interface builder for Smalltalk/V Windows

WindowBuilder, from Acumen Software, is a User Interface Management tool which greatly facilitates the rapid development of Smalltalk/V Windows applications. As its name implies, WindowBuilder enhances developer productivity by providing a "construction set" tool with which to interactively design application windows and dialogs in a "what you see is what you get" manner. Once you are satisfied with your design, WindowBuilder creates a new class to encapsulate your design, generating the Smalltalk methods which bring it to life.

At a list price of $149.95, this is a potent rapid application development tool which should be included in any Smalltalk/V developer's environment. Though there is room for improvement, this initial release of WindowBuilder is a much needed enhancement to Smalltalk/V Windows.

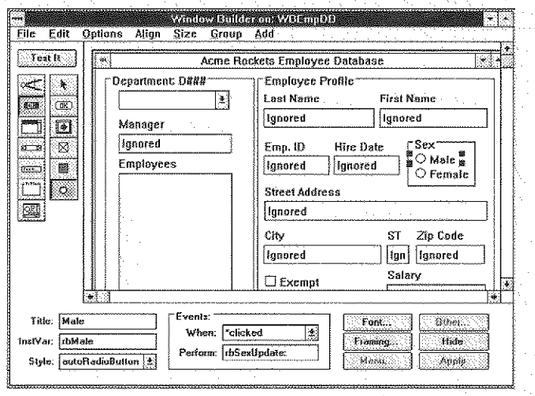


Figure 1. WindowBuilder tool building a database application window.

HOW DOES WINDOWBUILDER WORK?

WindowBuilder consists of software and a ninety-five page manual. The WindowBuilder tool and its associated classes are easily installed by filing in a single Smalltalk source file. Thirty-one classes are added to the base Smalltalk/V Windows environment. Some of these classes implement the WindowBuilder tool itself, but many are refinements and enhancements to the base system's window user interface Control classes. In addition to new classes, Acumen has made a significant number of modifications to methods in the base Smalltalk/V Windows classes.

Once filed in, a WindowBuilder menu is added to your Transcript window menubar giving you quick access to creating new and editing existing WindowBuilder windows and dialogs. WindowBuilder defines a new abstract class, WBTopPane, from which new windows and dialogs subclasses are created.

Figure 1 shows WindowBuilder in use to create a relatively complex application window. To place the **Male** RadioButton in the **Sex** GroupBox, as shown, the tool palette on the upper left side of the WindowBuilder window is used first to select a primary icon to place "Button" objects, after which a "RadioButton" secondary icon is selected. A crosshair cursor then appears to target the button's placement in the GroupBox.

The newly placed button displays "selection handles" to indicate that it is the active object. The Attributes Pane along the bottom of the WindowBuilder window is used to specify a default title, associated instance variable and Windows-specific style attributes. The Events group includes a **When** ComboBox which allows you to choose events to which the selected object will react. In this case, the RadioButton associated with the **rbMale** instance variable will react to a **clicked** event by sending its parent window the **rbSexUpdate** message. The Events group can be used to specify as many **when:** *event* **perform:** *method* associations as required by your design.

A well-implemented group of alignment options make it easy to create a clean window or dialog design. The **Distribute Horizontally** and **Distribute Vertically** options, which space objects evenly between two outermost selected objects are particularly useful and relatively rare in user interface design tools.

A Framing Parameters Editor is provided to specify the complex relationships among window control objects when the window is resized. In Figure 2, the Framing Parameters Editor is being used to specify that the upper-left corner and bottom of a ComboBox are fixed relative to the Parent window's top left dimension while its right dimension is scaled to the window's new size.

A Menubar Editor makes is easy to design dropdown menus to be added to your window designs. As with your basic window or dialog design, WindowBuilder generates the often complex and error-prone source to the methods which create and initialize your menus.

Working in concert, the tools provided by WindowBuilder make quick work of designing a window or dialog. As an interactive tool, you can save your design to its own WBTop-

Greg Hendley and Eric Smith

Giving application windows dialog box functionality in Smalltalk/V PM, part 1

Welcome to the first installment of what we hope will be long-running column! Smalltalk has been around for some years now. When Smalltalk was young, the idea of applications having windows the way cats have kittens was a new one. Smalltalk environments of yore preceded the proliferation of standardized window environments. Therefore, they tended to carry their own windowing system with them. These old clunkers would grab the whole machine (keyboard, screen, and mouse) and have their own way with them.

Of late, however, the world has been changing. For nearly every kind of desktop workstation, from the PC-clone to the top-of-the-line UNIX workstation, there is a standard windowing system available. Applications that run on these machines are increasingly expected to conform to the interface standards of the host windowing system. Further, they are expected to work with other applications running under the same windowing system.

Fortunately, Smalltalk has kept up. Both of the major vendors are beginning to support "host windows." In this column, we'll be providing information on the nuts and bolts of getting applications going in Smalltalk while working with the facilities provided by the host windowing system. To begin this issue, we'll dive right in to a two-part examination of how to build dialog boxes wholly within Smalltalk/V PM.

Dialog boxes are useful for displaying messages and gathering input from the user. In Smalltalk/V PM, there are two subclasses of DialogBox to handle simple cases. MessageBox is useful for getting quick yes/no or confirm/cancel information from the user. Prompter is useful for posing a question and soliciting an answer. There are other DialogBox subclasses for find and replace, choosing fonts, and defining new subclasses. In each case, a specific Presentation Manager (PM) dialog resource is used.

The resource defines the types and locations of the dialog's controls. To define a dialog with a different layout of controls, a new PM dialog resource must also be defined. This can be done with the dialog box editor or the linker and resource compiler that come with the Presentation Manager development kit.

If you have been using Smalltalk for a while you may ask, "Why can't all the work be done in Smalltalk?" This column proposes one approach to building custom dialogs wholly within Smalltalk. This approach creates a subclass of ApplicationWindow and gives it some useful behavior currently found only in DialogBox. In addition to convenience, there are two advantages to building dialogs wholly within Smalltalk. One advantage is you can use your own custom panes in addition to control panes. (DialogBox is restricted to holding control panes.) The other advantage is that once you know how, you can add the behavior to any application window.

ESSENTIAL BEHAVIORS OF DIALOGBOX

Since we will be taking the essential behavior of DialogBox and adding it to ApplicationWindow, let's identify what that behavior is. Under DialogBox in the encyclopedia of classes[1] is the comment:

> *"A DialogBox is a popup window used to display messages and gather input from the user. A dialog box can be modal or modeless. A modal dialog box requires that the user terminates that dialog box before using the window that opened the dialog. A modeless dialog box allows the user to continue to use the window without terminating the dialog box."*

So, an optional behavior of dialogs is being modal. (Application windows are modeless.)

You may have noticed another behavior: dialogs seem to stick with the application window that created them. If an application window and it's dialog are partially obscured by other windows and either the application or its dialog is selected, the application and the dialog window come to the front together. This sticking together is one of a set of behaviors dialogs have because of the ownership relationship between a dialog and its application window. The application window is said to own the dialog.

The option of being modal and the ownership relationship are considered to be essential behaviors of DialogBox. Other behaviors such as displaying messages, gathering user input, opening, closing, and passing messages are already part of being an application window. The rest of this column will discuss modality and ownership, where they are documented, what they mean, how Smalltalk/V PM uses them, some ways for you to use them, and finally, how to put it all together to make your own dialogs wholly within Smalltalk/V PM. The remainder of part 1 will cover modality. Part 2 will cover ownership and putting it all together.

MODALITY

Chapter 19, Dialog Windows, of ref. 2 (pp. 247–262) describes two kinds of modality dialogs may have in PM. A dia-

log may be system modal or application modal. When a dialog is system modal, the dialog takes control from all other windows in the system. When a dialog is application modal, it takes control from all other windows in the application. (As an aside, any window may be created system modal. Further discussion on this is deferred to a later issue.) Dialogs in Smalltalk/V PM are neither.

Dialogs in Smalltalk/V PM are modal only to the window that was active when the dialog was opened. As a result, modality for Smalltalk/V PM dialogs is handled within Smalltalk. This makes it fairly easy to move the modality behavior to ApplicationWindow.

The mechanism for making dialogs modal is documented on page 468 of the *Smalltalk/V PM Handbook*.[1]

> *"Dialog boxes can be made modal to the currently active window by putting self processInput as the last line in your dialog box's open method. processInput will not return until the user closes the dialog box (actually, until another method in your dialog box class sends self close). Again, see NewSubclassDialog for an example."*

> "... dialogs seem to stick with the application window that created them. If an application window and its dialog are partially obscured by other windows and either the application or its dialog is selected, the application and the dialog window come to the front together."

To understand what goes on when dialogs are made modal, let's look at the method processInput in DialogBox. This method is inherited by NewSubclassDialog:

```
processInput
        "Make the receiver modal to its owner window.
        This method doesn't return until close has been
        sent to the receiver."
    | cursor |
    Processor currentProcessIsRecursive ifTrue: [
        self error: 'Cannot do modal dialog during recursion.'].
    owner disable.
    cursor := Cursor.
    CursorManager normal change.
    sem := Semaphore new.
    [CurrentProcess makeUserIF. Notifier run] fork.
    sem wait.
    CurrentProcess makeUserIF.
    cursor change.
```

Two actions are taken to make the dialog modal: the owner is disabled and processing in the method is blocked. Disabling the owner is easiest, so let's look at it first.

DISABLING

Disabling the owner means the dialog's application window is prevented from receiving any more keyboard or mouse input. Conceptually, the dialog's owner is the application window that created the dialog. In implementation, the owner is set to the frame window of the active window when the dialog receives the message fromModule:id: or fromResFile:. The code that finds and sets the owner is the same in both methods. Examining the code confirms that the owner is a window handle.

```
owner isNil ifTrue: [
owner := Notifier activeMainWindow.
owner notNil ifTrue: [owner := owner frameWindow]].
owner isNil ifTrue: [owner := WindowHandle queryActive].
```

The method for disable in WindowHandle sends the method enableWindow:fEnable: to PMWindowLibrary, the sole instance of PMWindowLibraryDLL. In response, PMWindowLibrary calls the MS OS/2 function WINENABLEWINDOW. MS OS/2 responds by disabling the frame window and all its child windows (see pp. 260–261 of ref. 3). So, the owner ignores all future keyboard and mouse input until it is enabled. The dialog enables its owner in the method close.

BLOCKING

Blocking means that the Smalltalk process executing the method stops. The method does not return (and so the calling method does not continue) until the process is unblocked. The blocking is done in three lines:

```
sem := Semaphore new.
[CurrentProcess makeUserIF. Notifier run] fork.
sem wait.
```

The first line is simple. It initializes the semaphore. The second line makes a new user interface process and starts it processing events. The third line actually blocks the process the method is executing in. The process is blocked and the method does not return until the semaphore is signaled. The dialog signals its semaphore in its method close. Once the semaphore is signaled, the method processInput is resumed, the user interface process is restored, and the method returns. The method close for DialogBox closes the dialog and undoes both actions taken in processInput:

```
close
        "Close the receiver."
    owner enable.
```


of large, corporate sponsors that can afford to commit people and financial resources to its success.

Perhaps, today, the environment is ripe for such an organization. There are now numerous large corporations that are making strategic commitments to Smalltalk. These are the organizations that really need and can afford to support a Smalltalk users group and conference. My final remarks are to my colleagues in these organizations.

You and your organization have made a commitment to Smalltalk. Its future success is critical to your future and success. This requires a dynamic, vibrant community of Smalltalk users. Take control of your future, get involved and organized. Put together an organization, sponsor conferences and workshops, encourage standards. You know who you are, you know you have the need. So do it. If you don't know who your counterparts in other corporations are, then call me at (503-242-0725) and I will get you connected. Let's make Smalltalk succeed! ❁

Allen Wirfs-Brock can be reached at Instantiations, Inc., 921 SW Washington, Ste. 312, Portland, OR 97205, or by phone at (503)-242-0725.

MESSAGES

Allen Wirfs-Brock

Smalltalk, organization, and you

A forum for sharing ideas, tips, and experiences or just a place to have your say ...

This morning, I was reading through the papers in the conference proceedings of the 1991 USENIX C++ Conference. This is a collection of eighteen papers relating to the application, implementation, and possible extension of C++. In general, the papers are well written, most are informative, and some are controversial. Collectively, they show that there is a a large and thriving C++ community that is not only actively applying and evolving their language but also communicating, sharing, and recording their shared experiences. They clearly show that C++ is a living, dynamic language.

As a Smalltalk user and implementor, my immediate reaction to this collection was a sense of longing for a similar set of papers concerning Smalltalk. I know there is comparable work being done in the Smalltalk community. If this wasn't the case, Smalltalk would be a dead language. The problem is that there is currently no forum for Smalltalk users and developers to get together and share this work. Why not?

For several years, OOPSLA provided such a forum. If you look at the proceedings of the first two or three OOPSLA conferences, you will find a large number of papers that directly relate to Smalltalk. This is not the case today. Why? Because OOPSLA is now a large, formal, scientific conference that addresses all aspects of object-oriented technology. To be accepted at OOPSLA, a paper must address original ideas that are broadly applicable to object-oriented technology. A paper that is of utility only to the users of a particular language will normally not be accepted. At most, one or two Smalltalk-related papers will now be selected for an OOPSLA conference. Generally, the same will be true of C++ or any other language. This does not mean that only one or two good papers exist, but to publish more would result in an unbalanced conference. If eighteen of the twenty-three papers at this year's OOPSLA were the papers from the USENIX C++ Conference, only C++ programmers would attend OOPSLA.

The obvious solution is that we need a Smalltalk conference. This is not a totally new idea; others have suggested it in the past, but nothing has happened so far. Why? It's easy for an individual such as myself to get excited about the idea. I know what has to happen. I lived through the organization of the first OOPSLA. I could get on the phone and start calling people to get them involved... but wait, reality starts to kick in. Organizing a conference takes a lot of work and entails considerable financial risk. I run a small business. Can I afford to take the time away from my clients and employees? Could I carry the financial burden? Well, it was a nice idea, but back to work.

What is really necessary for the organization of a successful conference is an organization to back it. For OOPSLA, this was the ACM. For the C++ Conference, it is USENIX. USENIX describes itself as follows:

> *"The USENIX Association is a not-for-profit organization of those interested in UNIX and UNIX-like systems. It is dedicated to fostering and communicating the development of research and technological information and ideas pertaining to advanced computing systems, to the monitoring and encouragement of continuing innovation in advanced computing environment, and to the provision of a forum where technical issues are aired and critical thought exercised so that its members can remain current and vital. To these ends, the Association conducts large semi-annual technical conferences and sponsors workshops concerned with varied special-interest topics..."*

> A successful users group must be a response to a real need, a "pull" from the user community.

What about Smalltalk? Unfortunately, there is no comparable Smalltalk user's organization. Past efforts to create such an organization have been unsuccessful. Like conferences, user organizations take a considerable investment of time and money. Past efforts were "pushed" by vendors or individuals who had neither the time nor financial resources to be successful. A successful users group must be a response to a real need, a "pull" from the user community. In addition, it must have the backing


```
    owner makeActive.
    Notifier remove: self.
    PMWindowLibrary destroyWindow: handle.
    sem notNil ifTrue: [
        sem signal.
        Processor terminateActive.]
```

The first two lines enable the owner (the frame window of the dialog's application window) and restore it as the active window. The next two lines do some clean-up necessitated by the dialog having been created differently than a normal window. The last three lines signal the semaphore and kill the active process. Signaling the semaphore allows the process that was waiting on the semaphore to resume. This lets the method processInput resume and return.

ADDING MODALITY TO APPLICATIONWINDOWS

This is where you try out what you just read. Start by creating a subclass of ApplicationWindow. Give it an instance variable to hold the semaphore:

```
ApplicationWindow subclass: #DialogApplicationWindow
    instanceVariableNames: 'sem '
    classVariableNames: ''
    poolDictionaries: 'PMConstants '
```

Copy the method processInput from DialogBox. At the end of processInput add the line:

```
    super close. "<<Added because of difference between close in
DialogBox and ApplicationWindow."
```

Copy most of the method close from DialogBox. Modify it to look like this:

```
close
        "Close the receiver."
        "Most of this is copied from
        DialogBox. GLH 18 July 1991."
    owner enable.
    owner makeActive.
    sem notNil
        ifTrue: [
            sem signal.
            Processor terminateActive.]
        ifFalse: [super close.] "<<Added in case I am not modal. "
```

The differences between the close methods in DialogBox and ApplicationWindow will be covered in the next installment of this column. For now, just trust that this is necessary. The class will also need a method for finding and setting the dialog's owner. Make the three lines from from Module:id: into a method:

```
findAndSetOwner
        "Find the active window and make it my owner.
        This code is copied from DialogBox>>fromModule:id:.
        GLH 18 June 1991."
    owner isNil ifTrue: [
        owner := Notifier activeMainWindow.
        owner notNil ifTrue: [owner := owner frameWindow]].
    owner isNil ifTrue: [owner := WindowHandle queryActive].
```

The last method is for convenience:

```
openModal
        "Open and become modal the way
        most dialogs do. GLH 18 June 1991."
    self findAndSetOwner.
    super open.
    self processInput.
Now, test the class. Open a workspace. This will be the application
window for the dialog. From the workspace type, select, and do
together:
Temp := DialogApplicationWindow new.
Temp openModal.
Terminal bell.
```

Notice you can no longer type or use the mouse with the workspace. Also, note the last line did not execute. The dialog is modal to the workspace. The workspace is disabled and the process is blocked. Now, close the dialog. The bell will sound. Closing the dialog unblocks the process as expected. Also, note the workspace is enabled so the mouse and keyboard work with it.

Note: if you completely covered the dialog with the workspace, all is not lost. Simply type and do

```
Temp close
```

from the Transcript. This is why the global variable Temp was used.

In part 2, we will add ownership to DialogApplicationWindow and tie everything together. If there is space, you'll be shown some other ways to use these dialog behaviors. ❁

REFERENCES

[1] *Smalltalk/V®PM Tutorial and Programming Handbook*, Digitalk Inc., Los Angeles, CA, January 1989.

[2] *Microsoft Operating System/2 Programmer's Reference*, Vol. 1, Microsoft Press, Redmond, WA, 1989.

[3] *Microsoft Operating System/2 Programmer's Reference*, Vol. 2, Microsoft Press, Redmond, WA, 1989.

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

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

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

COMPRESSING CHANGES IN /V WINDOWS

Charles-A. Rovira

After suffering through a series of embarrassing crashes, I came to the conclusion that the SystemDictionary>>compressChanges method for Digitalk's Smalltalk/V Windows and Smalltalk /V Mac lacked a little something in the robustness column. Here's what I did to remedy the situation.

SMALLTALK WILL CRASH IF ABUSED

As loath as anyone might be to admit it, Smalltalk does suffer from certain problems when dealing with resources that are not its own. The Mac and Windows are great environments when it comes to providing objects and functionality, but the integration with Smalltalk is not as complete as it needs to be. The holes are quite deep enough to break an ankle when you stumble into them.

A HANDLE IS NOT A MONIKER

It is possible to leave handles or pointers to objects lying around and to trip over these objects, handles, or pointers when saving the image or during the course of execution. Leaving things unreclaimed is easy to do during the heat of a debugging session ... I've done it often enough. Saving the image after every victory, no matter how minor it might seem, is an essential component of debugging, but save your image with unreleased handles and life will rapidly become unpleasant.

After a while, an image becomes slow, bloated, and unreliable. What could be better than starting fresh with a new copy, straight from the shrink-wrapped diskettes, and filing in all of your work. It's all been preserved in the change.log or the /V Mac image data fork. Unfortunately, the change.log contains a record of **everything** that you've done while developing your system of application. Every successful *doIt* execution, two copies of every class definition, every method you've defined, as many times as you've defined it, every selector you've gotten rid of. Everything!

Filing in your change.log will take the maximum amount of time and it's not likely to work. The *doIts* are the show stoppers. Any *doIt* that brought up a window or otherwise interfered with the scheduling process is likely to stop the filing in of the log.

CLEAN UP YOUR ROOM

There is a way to shrink the change.log, remove all of the miscellany, folderol, and failure, and leave only the shiny new code: Smalltalk compressChanges. As it comes shipped by Digitalk, this copies the code to a new change.log, adjusting all pointers as it does so, saving the image when it's finished, and throwing away the old change.log. Try it ... you'll like it. Except that we're attempting to dispose of a flabby, flatulent, or faltering image, so we're going to try filingIn the change.log, appropriately renamed, into a brand spanking new copy of /V ... ***Wrong!*** CompressChanges cleans up a little too much. All the class definitions are now missing. Filing in the change.log will halt at every class asking if you want to declare <yourClassNameHere>. Then it will merrily reject all the code because, since <yourClassNameHere> is not a class, it does not understand *methods* (don't ask ...). Also, classes that need initialization will once again need initialization and there will be nothing to tell you which these are. The last straw is that any global variables you might have used in your application are now in lost in dataspace.

WHAT K-TEL HAS TO TEACH US

Get one now! It's new! Improved! Get one NOW! It won't rust, rot, or testify in court! Get one now! It's tanker bilge. It's a dessert topping! It's both! Rated X, the unknown. Positively no one admitted. Consult your local listings.

GET A LOAD OF THIS

In keeping with the tradition of writing frightfully incomplete articles, there is one minor component missing from the listing included, mostly because it deserves a separate article in its own right. To move global variables and their values out of the image and onto a file from which they can be recovered requires something called a Loader. Due to idiosyncrasies peculiar to each implementation of Smalltalk, Loaders tend to be as individual as the system in which they reside. Also, since this facility is needed in a development environment, loaders, at least as I implement them, tend to use the compiler because operation is faster than using becomes:.

SON OF COMPRESSCHANGES

The code in Listing 1 is capable of compressing the change.log into a form that can be filedIn into a new image. After a short initialization sequence to record what classes and globals are in the new system, all of the classes and global variables are

This has led us to concentrate on developing documentation and figuring out how best to describe the system. One thing that we have learned is to concentrate on the big picture and ignore information that can be learned just as easily with the browser. Thus, pictures that list the entire class hierarchy are not important, but descriptions of the meaning of the hierarchy are. Lists of all the methods in a protocol are not important, but descriptions of what each method does are.

Although most of the work on TS has been done at the University of Illinois, Justin Graver, who did the original work on type inference, is now at the University of Florida and has several students working on projects related to the compiler. Thus, TS is a multiinstitution project. We hope that it will become reliable enough to be useful in the near future and that many more people will start to use it.

SMALLTALK CODE ARCHIVE

The University of Illinois has an archive of Smalltalk software and of papers on object-oriented programming. TS is not in the archive yet. However, the archive contains a lot of software that was developed at Illinois including Foible, a framework for visual programming environments that was written in Smalltalk-80. It also contains the archive of Smalltalk-80 developed by Manchester University software and the archive of Smalltalk-V software developed by the International Smalltalk Association.

You can access the archive by anonymous ftp to st.cs.uiuc.edu (which is currently an alias for speedy.cs.uiuc.edu at [128.174.241.10]) or by sending e-mail to archive-server@st.cs.uiuc.edu of the form

```
To: archive-server@st.cs.uiuc.edu
Subject:
path  yourname@your.internet.address
archiver shar
encoder uuencode
help
encodedsend ls-lR.Z
```

which will cause the archive server to e-mail instructions to you. Report problems with the archive to archive-manager@st.cs.uiuc.edu.

As a last resort, you can get the entire contents of the archive on an Exobyte tape or 1/4" QIC-24 (DC600A cartridges) in tar format, on Macintosh disks, or on DOS 3 1/2" inch disks by sending $200 to William Voss at Department of Computer Science, 1304 W. Springfield, Urbana, IL 61801. ❁

Ralph Johnson is in the Department of Computer Science at the University of Illinois at Urbana-Champaign. He can be reached there at 1304 W. Springfield, Urbana, IL 61801, or by phone at (217) 244-0098, or via e-mail at johnson@cs.uiuc.edu.

LAB REPORT

Ralph Johnson

The Typed Smalltalk project at the University of Illinois

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

The Typed Smalltalk project is one of several object-oriented projects at the University of Illinois at Urbana-Champaign, and the largest that uses Smalltalk. The goal of the Typed Smalltalk project is to make Smalltalk as fast as any other language by using optimizing compiler technology. We want to make Smalltalk fast without losing any of its advantages or changing the way it is used. We want to hide the compiler from the programmer and keep the programming environment just as interactive and useful for prototyping as Smalltalk has always been.

Typed Smalltalk is a large project with many components. These components fall into two categories, language changes and the compiler. The major language change is a type system that was designed to fit the way Smalltalk programmers program, not to force programmers to use a particular style. Type information does not change the meaning of a program but is just an annotation on an untyped program. Although the original motivation for the type system was to provide information that the compiler could use to make programs faster, it is also useful documentation.

One important part of the type system is a type inference system that automatically finds the types in a program. The compiler can infer a type for a method (the types of all the variables used in the method and its return type), but a programmer can refine these types to make the type more precise. The goal is for the programmer to rely on type inference when a program is being written and is changing a lot, and then to narrow down the types as the program moves from development to production use.

The compiler (TS) uses type information to convert Smalltalk into efficient machine code. TS is entirely written in Smalltalk. It has been designed to be portable and has a table-driven code generator. We currently have code generators for the M68020, the NS32032, and the SPARC and are working on one for the i80386.

The biggest problem with the back-end is that it is slow. The best way to solve this is to compile it. Unfortunately, TS does not work well enough yet to compile itself.

The project has had two major problems. We are using a single technique to attack both problems. The first problem is endemic to building large software systems: making the system reliable. The second is endemic to academic projects: building a large system on a shoestring budget with high attrition rates of workers. Although we have had some funding from NSF and a little from Tektronix and BNR, a lot of the work on the compiler has been done by unsupported students working on thesis projects. These volunteers work for the fun of it, so the work must be fun, and they tend to leave just about the time they have mastered the system.

The computer center of my alma mater had a sign that gave the "ten laws of computing." I don't remember most of them, but one of them was that "All nontrivial programs have bugs in them." A corollary was "If your program has no bugs then it is trivial."

Since we are trying to make a reliable optimizing compiler, this implies that we must build a trivial optimizing compiler. Unfortunately, optimizing compilers are big and complicated,and tend to be buggy. In spite of this, we have tried to make TS as simple as possible by rewriting parts that are complex. We have rewritten some of the parts at least a half dozen times. This has greatly improved the reliability of TS, though there are many parts that are still complex and TS is still unreliable. Part of the Smalltalk culture is rewriting code until it is elegant, easy to understand, and reusable. Our strategy fits into this culture perfectly.

Another reason for reliability problems is improper testing. Most people do not think that testing is fun, so volunteers are unlikely to develop and implement thorough test plans. Also, exhaustive tests of optimizing compilers are very difficult. Finally, the Smalltalk culture does not recognize the value and difficulty of testing and there are few tools to support it. Although the first two problems are peculiar to us, the third is widespread in the Smalltalk community and needs to be fixed.

One of the keys to having thesis projects produce useful software is to limit the scope of each project and to give the student time to rewrite the software several times. This not only produces better software, but the students are happier because they know they have done a good job. A good MS thesis project is to rewrite an overly complex part of the compiler, so this approach helps make the compiler more reliable.

MS students tend to spend a semester learning TS and Smalltalk, a semester doing useful work, and a couple of months writing a thesis. The high attrition rate has made painfully obvious the need for high-quality documentation.

defined, the methods are saved, and the required class initialization is performed and global values are loaded in.

To keep track of what additions or changes have been made to the original image it is necessary to determine what classes and globals are present in the image. This is the function of the initialization sequence. The code not directly related to *SystemDictionary>>compressChanges* is there to keep track of the system as it changes.

- *Class>>comment* is necessary because I use class comments for a class hinting mechanism that allows me to verify methods to ensure that I won't get **'does not understand:'** walk-backs. It can also ensure that cloned images don't contain more objects, classes, or methods than is absolutely necessary. Like the Loader, this deserves its own article.
- *Class>>removeFromSystem* was modified to add the very last line. It also checks if the class has instances and prompts the operator through a ConfirmDialog. This is a standard Widgets/286, Widgets/Mac dialog which I implemented in /V Windows to maintain compatibility. The method can be changed to simply abort if there are any instances of the class.
- *Class>>removeInstances* just does what it says it does.
- The following methods have been modified to keep comments around across recompilation: *Class>>subclass:instanceVariableNames:classVariableNames:poolDictionaries:*, *Class>>variableByteSubclass:classVariableNames:poolDictionaries:*,*Class>>variableSubclass:instanceVariableNames:classVariableNames:poolDictionaries:*
- *MetaClass>>name:environment:subclassOf:instanceVariableNames:variable: words:pointers:classVariableNames:poolDictionaries:comment:changed:* was modified to remove redefined classes from a list of the classes that were present in the original image.
- *SystemDictionary>>compressChanges* is where the metaphorical rubber hits the yellow brick road. It has been modified to add a preamble to the image that gathers all classes and all global variables defined in the original image and to do messages sends of the following selectors:
  - *SystemDictionary>>compressClassDefsOf:into:* places all new or changed class definitions into the change.log. This method is implemented recursively to ensure that the class hierarchy is respected. To make it easier to relate class definitions with their order in the CHB, the subclasses are sorted alphabetically. This message is sent immediately after the preamble.
  - *SystemDictionary>>compressClassInitsInto:* finds all user-defined classes that implement an initialize selector and places the appropriate message send so that the class will be initialized automatically on filing in the log.
  - *SystemDictionary>>compressGlobalDefsInto:* reserves name space for all new globals used in any methods in the change.log. This message is sent immediately after having saved all of the global values.
  - *SystemDictionary>>compressGlobalInitsInto:* loads the globals from a 'recovery.dat' file. This will be the last message in the change.log
  - *SystemDictionary>>compressGlobalValuesInto:* saves all new globals used in the image in a 'recovery.dat' file. This is sent immediately after saving all class definitions.
  - *SystemDictionary>>removeKey:ifAbsent:* keeps track of deleted globals. If if it necessary to modify globals that come with the system, remove them from the system before replacing them with their new value. This ensures that they are unloaded.

> **Due to idiosyncrasies peculiar to each implementation of Smalltalk, Loaders tend to be as individual as the system in which they reside.**

PITFALLS

There are none. I have used this method to successfully save change.logs that contained all information necessary to recover my system after some real doozies. I sometimes refresh the image and fileIn the change.log to ensure that I have no obscure semicircular references or other uncollected garbage.

Since implementing these changes, I am much more comfortable about experimenting with objects and resources outside Smalltalk's control. When I'm trying a triple somersault from the flying trapeze, it's always nice to have a safety net. ❁

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

Listing 1.

```
Class methods
comment
    "answer the class comment"
    ^comment
removeFromSystem
    "Remove the receiver from Smalltalk. Report an error if there are any sub-
    classes or instances of the receiver."
... added line of code
    OriginalClasses remove: myName asSymbol ifAbsent: []
removeInstances
    self withAllSubclasses do: [:aClass |
        aClass allInstances do: [:anInstance |
                anInstance become: String new] ].
subclass: classSymbol
    instanceVariableNames: instanceVariables
    classVariableNames: classVariables
    poolDictionaries: poolDictNames
    "Create or modify the class classSymbol to be a subclass of the receiver with
    the specified instance variables, class variables, and pool dictionaries."
... inserted lines of code
    | aComment originalClass |
    comment := String new.
    originalClass := Smalltalk at: classSymbol ifAbsent: [].
    originalClass notNil ifTrue: [
        aComment := originalClass comment].
... modified line of code
        comment: aComment
        changed: nil
variableByteSubclass: classSymbol
    classVariableNames: classVariables
    poolDictionaries: poolDictNames
    "Create or modify the class classSymbol to be a variable byte subclass of the
    receiver with the specified class variables and pool dictionaries."
... inserted lines of code
    | aMetaClass aComment originalClass |
    aComment := String new.
    originalClass:= Smalltalk at: classSymbol ifAbsent: [].
    originalClass notNil ifTrue:
        [aComment := originalClass comment].
... modified line of code
        comment: aComment
        changed: nil

variableSubclass: classSymbol
    instanceVariableNames: instanceVariables
    classVariableNames: classVariables
    poolDictionaries: poolDictNames
    "Create or modify the class classSymbol to be a variable subclass of the re-
    ceiver with the specified instance variables, class variables, and pool dictio-
    naries."
... inserted lines of code
    | aMetaClass aComment originalClass |
    aComment := String new.
    originalClass := Smalltalk at: classSymbol ifAbsent: [].
    originalClass notNil ifTrue: [
        aComment := originalClass comment].
... modified line of code
        comment: aComment
        changed: nil
MetaClass methods
name: newName
    environment: aSystemDictionary
    subclassOf: superclass
    instanceVariableNames: stringOfInstVarNames
    variable: variableBoolean
    words: wordBoolean
    pointers: pointerBoolean
    classVariableNames: stringOfClassVarNames
    poolDictionaries: stringOfPoolNames
    comment: commentString
    changed: changed
        "Private – Create or modify the class and the metaclass of name new-
        Name to be as defined by the arguments. Check if an OriginalClass is
        being redefined"
... added line of code
    OriginalClasses remove: newName asSymbol ifAbsent: [].
    ^answer
SystemDictionary methods
compressChanges
    "Build a new change log file retaining only the latest version of changed
     methods in the current change log. Save the image to the image file."
    | logDirectory stream tempLogName dialog aFileStream |
    dialog := DialogBox new
        fromDLLFile: 'vwdlgs.dll'
        templateName: 'CompressingChange',
    dialog showWindow.
    logDirectory := (Sources at: 2) file directory.
    stream := logDirectory newFile: 'ChangLog.tmp'.
    stream lineDelimiter: Cr.
    tempLogName := stream pathName.
... added lines of code
    stream nextPutAll:
' "evaluate"
    | originalClasses originalGlobals |
    originalClasses := SortedCollection new.
    originalGlobals := SortedCollection new.
    Smalltalk associationsDo: [:each |
        (each value isKindOf: Class)
                ifTrue: [originalClasses add: each key ]
                ifFalse: [originalGlobals add: each key]].
    Smalltalk at: #OriginalClasses put: originalClasses.
    Smalltalk at: #OriginalGlobals put: originalGlobals.!!';
    cr.
... added lines of code
    self compressClassDefsOf: Object into: stream.
    self compressGlobalValuesInto:
        (aFileStream := File pathName: 'recover.dat').
    aFileStream close.
    self compressGlobalDefsInto: stream.
... added lines of code
    self getSourceClasses do: [ :class |
        self compressChangesOf: class class into: stream.
        self compressChangesOf: class into: stream].
... added lines of code
    self compressClassInitsInto: stream.
    self compressGlobalInitsInto: stream.
... added line of code
    stream close.
    (Sources at: 2) close.
    File remove: (Sources at: 2) pathName.
    File rename: tempLogName to: (Sources at: 2) pathName.
    Sources at: 2 put:
        (logDirectory file: (Sources at: 2) file name).
    (Sources at: 2) lineDelimiter: Cr.
    ApplicationWindow new saveImageNoConfirm.
    dialog close
    compressClassDefsOf: aClass into: aStream
"Write into a stream all of the hierarchy of class definitions that are new to
the image."
    | classes |
    (OriginalClasses includes: aClass name asSymbol) ifFalse: [
        aClass fileOutOn: aStream.
        aStream nextPut: $!!; cr ".
        Transcript cr; show: aClass name"].
        classes := aClass subclasses asSortedCollection:
        [ :first :second | first name < second name].
    classes do: [ :aSubclass |
        self compressClassDefsOf: aSubclass into: aStream]
compressClassInitsInto: aStream
    "Write initialization code for all classes that have it"
    | initializedClasses |
    aStream nextPutAll: ' "evaluate" '; cr.
    initializedClasses := self select: [ :anEntry |
        (anEntry isKindOf: Class) &
          (anEntry class selectors includes: #initialize) &
          (OriginalClasses includes: anEntry) not].
    initializedClasses do: [ :aClass |
        aStream nextPutAll: self name , ' initialize.'; cr].
    aStream nextPutAll: '!!'; cr
compressGlobalDefsInto: aStream
    "Set up all global names"
    | globalsDictionary |
    (OriginalGlobals includes: #OriginalClasses)
        ifFalse: [OriginalGlobals add: #OriginalClasses].
    (OriginalGlobals includes: #OriginalGlobals)
        ifFalse: [OriginalGlobals add: #OriginalGlobals].
    aStream nextPutAll: ' "evaluate" '; cr.
    globalsDictionary := self reject: [ :anEntry |
        anEntry isKindOf: Class].
    globalsDictionary keysDo: [ :aSymbol |
        (OriginalGlobals includes: aSymbol) ifFalse: [
                aStream nextPutAll: ' Smalltalk at: #',
                  aSymbol printString, ' put: nil.'; cr]].
    aStream nextPutAll: '!!'; cr.
compressGlobalInitsInto: aStream
        "Get globals if we can load them."
    aStream nextPutAll: ' "evaluate"
    | collectionOfAssociations aFileStream |
    Smalltalk at: #Loader ifAbsent: [
        Transcript cr; show:
        "Loader not available. Globals not loaded.",
        ^nil].
    aFileStream := Disk file: "recover.dat".
    aFileStream size > 0 ifFalse: [
        aFileStream close.
        File remove: aFileStream pathName.
        Transcript cr; show:
        "Recover.dat not available. Globals not loaded ".
    ^nil].
    collectionOfAssociations := Loader new readFrom: f.
    aFileStream close.
    collectionOfAssociations do: [:aPair |
    Smalltalk add: aPair]. !!!!'
compressGlobalValuesInto: aStream
    "Save (unload) all of the globals into aStream"
    | aCollection classList |
    Smalltalk at: #Loader ifAbsent: [
        Transcript cr; show:
;               'Loader not available. Globals not saved.'.
    ^nil].
    aCollectionOfAssociations := OrderedCollection new.
    classList := #(Behavior Persistent ClassReader
            ClipboardManager Compiler Context CursorManager
            DelayedEvent DeletedfClass Directory Dos
            DynamicDataExchange EmptySlot File Font
            GraphicsMedium "GraphicsTool" InputEvent Loader
            Menu MenuItem Message NotificationManager
            ProcessScheduler ViewManager Window WinHandle
            WinInfo WinLogicalObject WinStructure).
    (OriginalGlobals includes: #OriginalClasses)
            ifFalse: [OriginalGlobals add: #OriginalClasses].
    (OriginalGlobals includes: #OriginalGlobals)
            ifFalse: [OriginalGlobals add: #OriginalGlobals].
    self associationsDo: [:aPair |
            (aPair value isKindOf: Class) ifFalse: [
              (OriginalGlobals includes: aPair key) ifFalse: [
                    (classList includes: aPair value) ifFalse: [
                      aCollectionOfAssociations add: ea ] ] ] ].
    Loader new write: aCollectionOfAssociations to: aStream.

removeKey: aKey ifAbsent: aBlock
    "We're getting rid of something in Smalltalk. Check if it's an
    OriginalGlobal."
    OriginalGlobals remove: aKey ifAbsent: [].
    ^super removeKey: aKey ifAbsent: aBlock
```

The Smalltalk Report

The International Newsletter for Smalltalk Programmers

January 1992 Volume 1 Number 4

SHOULD CLASSES HAVE OWNERS? PERSPECTIVES FROM EXPERIENCE

By S. Sridhar

This is a response to Juanita Ewing's "Should classes have owners?" article in the September 1991 issue of *The Smalltalk Report*. There are several themes in the article with which I'd like to take issue. I have been a Smalltalk programmer for some years now, and for about the last nine months several of us at Knowledge Systems Corp. have been extensively using a commercially available development environment that pervasively supports the concept of class ownership. This is the ENVY/Developer team development tool running on Smalltalk/V PM and Smalltalk/V Windows. This is a powerful programming environment designed to facilitate cooperative software development among a team of programmers. The tool is flexible enough to cater to the needs of multiperson teams as well as the lone programmer. For the purposes of this article, I shall use the term ENVY to refer to ENVY/Developer.

It is in the context of my experience of having developed Smalltalk code using a team tool like ENVY in an inherently multiperson environment that I shall address each of the issues Juanita has raised. I shall also attempt to provide technical as well as sociological answers to the questions she has raised. I use ENVY here to set a practical context for documenting my experience with many of the class ownership issues discussed in the original article. Readers should not misconstrue this as a commercial plug for the product.

TERMINOLOGY PRELUDE

Before delving into specific issues, let us define some key terms relevant to this discussion. ENVY supports the notions of *class owners* and *class developers*. A class is only one of many software components that have an ownership aspect associated with them. Ownership implies that someone is responsible for controlling a software component's evolution. This control manifests itself in the fact that only an owner can *release* a class for public consumption.

The granularity of a software component can be varied: a method, class, set of classes, set of sets of classes, etc. ENVY also supports an additional programming environment structure called an *application*. An application is a collection of defined and extended classes that together accomplish a well-defined purpose. In addition to providing a physical organization of related classes, it also serves as a large-grain reusable component. Team members no longer just talk about reuse of a single class; they talk about reuse of functionality. This is good because the responsibility for accomplishing a given piece of functionality may be distributed among a set of closely collaborating classes.

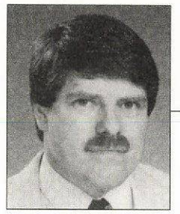
Class developers are team members who may author one or more classes in the application. They may be distinct from the person who actually owns the class.

WHO IS BEST QUALIFIED TO FIX THE BUG OR WRITE THE NEW METHOD?

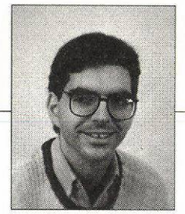
Juanita writes: "assume the owner of a class is rewarded for producing a reusable class. What if another developer finds a bug in that class or thinks of a useful extension? In a

continued on page 4...

Contents:



John Pugh



Paul White

EDITORS' CORNER

In last month's editorial, we urged you to take our columnists to task if you did not agree with their opinions on particular topics. Well, you did just that! The approach to change management proposed by Juanita Ewing in her opening Getting Real column, "Should classes have owners?," has spurred several well-known members of the Smalltalk community to put forward their ideas. In this month's lead article, S. Sridhar from Knowledge Systems argues that, based on his experience, class ownership is indeed a primary component of any strategy for managing change in large Smalltalk applications. Next month, Jeff McKenna will put forward his view that change management is best organized around what he refers to as the two distinct phases of software development using Smalltalk—functional expansion and consolidation. Change management seems to be a topical subject right now, and we look forward to hearing your views.

Two of our regular columnists appear in this issue. Rebecca Wirfs-Brock continues her Object-Oriented Design column by discussing the importance of understanding object roles and responsibilities. In this month's Getting Real column, Juanita Ewing begins a two-part article on the appropriate use of class variables and class instance variables. Also in this issue, Glen Reid, the architect of the Smalltalk/370 project, continues his description of their project. In this issue, he discusses in detail many of the implementation issues that are specific to implementation on a mainframe, including a scheme to introduce explicit variable typing in Smalltalk.

Rounding out this issue, Jon Hylands takes a look at the first of a new line of third party Smalltalk products, Profile/V, a code profiling tool that can be used to monitor the performance of Smalltalk applications. Finally, Dan Lesage reviews *Object-Oriented Modeling and Design* by James Rumbaugh et al.

The Smalltalk Report is still finding its feet. Let us know what you like, what you don't like, and what you would like to see. We look forward to hearing from you and hope you enjoy this issue.

John Pugh Paul White

The Smalltalk Report (ISSN# 1056-7976) is published 9 times a year, every month except for the Mar/Apr, July/Aug, and Nov/Dec combined issues. Published by COOT, Inc., a member of the SIGS Publications Group, 588 Broadway, New York, NY 10012 (212)274-0640.

Mailed First Class. Subscription rates 1 year, (9 issues) domestic, $65, Foreign and Canada, $90, Single copy price, $8.00. POSTMASTER: Send address changes and subscription orders to: The Smalltalk Report, Subscriber Services, Dept. SML, P.O. Box 3000, Denville, NJ 07834. Submit articles to the Editors at **91 Second Avenue, Ottawa, Ontario K1S 2H4, Canada.**

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 **Agorics Project** announced the opening of an **online Smalltalk Components and Consulting market** on AMIX, the new electronic marketplace for information provided by Autodesk, a subsidiary of the American Information Exchange Corp. (AMIX). In this market, Smalltalk users will be able to buy and sell classes, methods, tools, applets, and any other Smalltalk-related information. Users will also be able to offer and request Smalltalk consulting services. Features include email, negotiation facilities, listings of sellers' resumes and references, listings of comments on components by previous buyers, and more.

For more information, contact Howard Baetjer, The Agorics Project, 10364 Bridgetown Place, Burke, VA 22015; phone and fax (703) 250-4760; email agorics@gmuvax.gmu.edu.

InputForms is a program designed for the interactive development of input forms and all kinds of windows running under Windows 3.0 and Smalltalk/V Windows. Features include the ability to interactively select child controls and define size, position, brush, foreground color, background color, font, etc.

For more information, contact Vlastimil Adamovsky, 66 rue de Bourgogne, L-1272 Luxembourg; phone 352 420884.

Empower Software has announced the availability of the **Smalltalk Project Browser**, a source code management tool for Smalltalk/V Windows and PM systems that adds a powerful layer of control to the Smalltalk environment. It is also useful as a development shell from which other Smalltalk development tools are launched. The Smalltalk Project Browser provides support for code porting and maintenance across Smalltalk platforms, management of class dependencies, system integration, automated code documentation, and code distribution and packaging.

For more information, contact Empower Software, 9601 Wilshire Blvd., Ste. 1144, Beverly Hills, CA 90210.

Digitalk, Inc. has announced availability of a new release of its **Smalltalk/V PM** that gives software developers a jump start on developing new applications that take advantage of the power of IBM's upcoming version 2.0 of OS/2. In addition to enhanced features and power, Digitalk's Smalltalk/V PM 1.3 release includes support for IBM's Common User Access '91 (CUA) controls that are at the heart of IBM's new advanced OS/2 2.0 graphical user interface.

For more information, contact Barbara Noparstak, Digitalk, Inc., 9841 Airport Blvd., Los Angeles, CA 90045; (213) 645-1082; fax (213) 645-1306.

WHAT THEY'RE SAYING ABOUT SMALLTALK

Excerpts from industry publications

... Momenta built the [PenTop] machine around the object-oriented language Smalltalk. Everything in the PenTop's environment is an object, so users can link anything in the machine—from internal toolbox functions to their own sketches, text, and presentations—to one another. The machine runs all popular DOS and Windows applications, and will support Microsoft's PenWindows when it becomes available ...

Momenta Rewrites the Notebook Rules, Richard Doherty, ***Electronic Engineering Times****, 10/7/91*

... In addition to the visual orientation, there are two other reasons I'm attracted to Serius' product. One is the level of abstraction of the objects. Most object-oriented languages today are for professional programmers (e.g., C++ and Smalltalk) and that means the objects are at a relatively low level of abstraction to provide sufficient control for speed and memory efficiency. Serius Programmer, on the other hand, has very robust objects for an application generator ... The second reason I like the package is the relatively broad support for data types.

A Serius Approach to Programming, Rich Bader, ***PC Letter****, 9/16/91*

... Specialized OOP environments like Smalltalk tend to frighten programmers used to the procedure-oriented approach of traditional languages...Although embedding OOP technology in existing languages like Pascal or C has really boosted OOP, the tendency for programmers using those tools is to keep on doing things the same way, with only a few changes. There's still a big learning curve, and, if you give a C programmer a C++ compiler, he'll probably just write C code. It's hard to lose old habits ...

... [Ron Fisher says] "Smalltalk's concepts are very different, but once you can deal with them conceptually, you can write much better programs. Smalltalk is a whole environment, not just a language. To me, C++ is a kit car, and Smalltalk is an Acura NSX. C++ wasn't thought out thoroughly as an object-oriented language. It exists because C exists. You can do a lot more low-level stuff in C that you can with Smalltalk. C lets you get at the iron much better, but if it wasn't for C, C++ wouldn't have much of a following"...

Double Plus Good, Gordon McLachlan, ***HP Professional****, 9/91*

... But in a world increasingly jammed with OOP proselytes, we still don't have an OOP graphics front end for these [graphics] libraries. I would like to see something that would give me ONE Object Oriented Design perspective with support for several graphics libraries ...

Graphic Developer's Taste Test, William E. Gates, ***Midnight Engineering****, 10/91*

... The more advanced pen-computing operating systems use object-oriented design for memory management. In contrast to desktop GUI applications, which may require multiple megabytes of memory, object-oriented applications typically require only about 100K to 200K because the operating system conserves memory by eliminating redundant code ...

Is the Pen Mightier?, Kathleen Melymuka, ***12A-550 CIO****, 9/15/91*

... Building a single, integrated model for the problem domain is something the securities industry has to do. We're face to face with the complexity of the solution right now. Other industries won't be far behind. Take a close look at your own problem domain; you may find that the celebrated paradigm shift is not a problem of changing the way people think but of dealing with the resulting solution ...

The Complexity of the Solution, Bill Welch, ***Object Magazine****, 9-10/91*

... continued from p.17

slightly thrown off since the style of the other analysis and design chapters gave me much more concrete choices to make. And, since this is *The Smalltalk Report*, I can also say that the Smalltalk language is somewhat slighted as a potential choice for implementation language primarily because the authors refer to it as a weakly typed language. I believe that there exists confusion here between the use of *strong* typing and *static* typing. As every Smalltalk programmer knows, Smalltalk is a strongly typed language.

Overall, I highly recommend this book to anyone who is interested in learning more about OO analysis and design. It contains good, sound, practical knowledge drawn from real-world examples. The methodology is flexible, allowing its users to emphasize those modeling techniques that make sense in their shop, while deemphasizing those that are irrelevant. The book clearly gives a path that takes the modeler from known structured techniques and allows him to migrate this knowledge into the realm of OO analysis and design. In short, this book has something for everyone using or considering the use of OO technology. ■

Dan Lesage has been involved with object-oriented programming since 1986 and Smalltalk since 1988. Currently, he is the Project Manager, Turnkey Systems at Object Technology International in Ottawa, Canada. His current interests include distributed computing, data communications, and object-oriented analysis/design. He can be reached at Object Technology International, (613) 228-3535, or dan@oti.on.ca.

continued from page 1...

system with class ownership, the owner writes the code to fix the bug or writes a new method. He is the one motivated to make the class more reusable."

First, the case where a developer finds a bug. Suppose I own a reusable class called Drawing. If another developer, say Harry, finds a bug in Drawing, he creates a scratch edition of the application containing the class Drawing, creates a new edition of Drawing, fixes the bug, versions the change, and informs the owner via email or otherwise of the fix. I, as the owner, can examine the fix at my leisure, assess the impact on the clients of the method, and, if all is well, incorporate the fix into a future version of Drawing and then *release* it for public consumption. Alternatively, I could simply release the version of Drawing that Harry created. In the meantime, Harry can continue to use the scratch edition of Drawing and do anything he pleases to any of the existing methods of Drawing without impacting any other team member. When I have released a new version of Drawing, he can *load* it into his environment, replacing the scratch edition.

Thus, it is that Harry and I have resolved the bug by engaging in a harmonious electronic "conversation" without disrupting any other team member. He found the bug, submitted a fix, and continued to do his work with his fix without awaiting my approval. I, as the owner of the method, evaluate the quality of the fix, assess the impact of the fix, and then fold it into the next version of the class and release it for our team's use. The owner is the best person to assess the overall impact since he is the one who most intimately knows the raison d'etre for the method in the first place. He is probably the most aware about the way in which existing and potential clients use the method. ENVY automatically records the author and time stamp of the fixed method.

Alternatively, Harry can create a new working copy or *edition* of Drawing along a different stream of development or versioning branch. When he is done fixing the bug, he versions the class with a mnemonic version label. (The mnemonic label is not required; it is just a convention we have adopted to meaningfully identify the different versions of a class.) The owner then merges his contributions with the officially released version of Drawing. The point of all this is that:

- With good communications (which is required anyway for healthy project sociology), class ownership does not hamper the evolution of a class into the reusable club. This is primarily because changes to the class can be made asynchronously.
- The owner reviews the fix in a different context from that of the other developers. It is his responsibility to guarantee the proper functioning of all the advertised interfaces of his class and to the extent possible be familiar with all the usage contexts of his class.

ADDING CLASS EXTENSIONS

The case where Harry finds a useful extension to Drawing is easily dealt with in ENVY. As a matter of fact, this situation occurs constantly in our work with system classes like String, Stream, etc. ENVY provides a programming environment abstraction called *class extension* that allows a developer to add brand new methods to an existing class. These method extensions are localized to the application in which the extension is defined. Thus, Harry can add a new method to Drawing by creating an extension of Drawing in his application. Even though I am the owner of Drawing, Harry does not require my permission to add the useful extension he needs. Furthermore, this extension does not compromise the integrity of the original class. A malicious Harry could, of course, destroy the class' integrity by writing a method extension that corrupts the internal state of the class in a way that is incompatible with the rest of the class' behavior. The users of Harry's code are the losers. Team sociology being what it is, Harry would be quickly exposed by the users and be pressured to undo his mischief.

It should be noted that the person who creates a class extension in a different application actually owns the extension. Class extensions are a powerful mechanism for specifying and managing application-specific behaviors for existing classes and for dealing with orthogonal protocols for classes where several developers are authoring different parts of the same class. By splitting these orthogonal protocols along their functional views using applications, multiple developers on a single class can be managed realistically and effectively.

REWARDING REUSE

Juanita correctly notes that if a reusable class is provided by a team of developers then the entire team should be rewarded. It is our experience that a reusable class usually has a primary author (or owner in ENVY parlance) and it can have multiple developers different from the author. These secondary authors can be reviewers, bug finders and fixers, and maybe even coauthors. Again taking the Drawing example, I may find that Harry has made a dozen extensions to Drawing in his application. Upon close examination, I determine that these extensions are useful and general enough to warrant inclusion in my Drawing class. In ENVY, as the class owner, I simply add Harry as a developer of the class, have him *promote* the dozen deserving methods to my reusable rendition of Drawing. All the newly promoted methods carry Harry's imprimatur. Thus, Harry and I are established as coauthors of Drawing. Since the programming environment explicitly identifies the people who are working on an application (a large-grain reusable component), it is easy to identify who to reward. A picky manager can even measure the relative contributions to the reuse genre and can thereby dispense rewards proportionately!

There is an interesting sociological aspect to this reward issue that runs somewhat orthogonal to class ownership. If Harry makes a change to my class that I don't like—as in Juanita's world—who wins? As my colleague Lynn Fogwell

BOOK REVIEW

Reviewed by Dan Lesage

OBJECT-ORIENTED MODELING AND DESIGN

by J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen
Prentice Hall, Englewood Cliffs, NJ, 1991

This is the book to recommend to your MIS/DP customers that are considering the use of OOP in their company but don't know where to start down the path toward the Holy Grail. The investment in an OO language may be considered too risky for the average data processing manager, without knowing how OO can benefit his or her complete development cycle. In that regard, the DP manager will likely wish to understand the benefits of OO in terms of a formal methodology. Rumbaugh et al. describe their object modeling technique (OMT), which is a gentle mutation of existing structured analysis/structured design (SA/SD) methodologies plus entity-relationship (ER) diagrams into an OO one. Should your DP customer already be using structured techniques in his or her shop, this book will help ease the transition toward OO. It should be no surprise that a large part of OMT follows Rumbaugh's own work in combining objects with relations at GE, as described in several of the *OOPSLA Proceedings*.

The book consists of five major sections: motivation, modeling, methodology, implementation, and example systems. The motivation part covers the normal questions of why one would want to use OO techniques. The modeling section presents the components of the OMT techniques that are based on three diagramming techniques. Two of them are (hopefully) already being used by your MIS/DP customer: Harel state diagrams, which are used as the dynamic model, and data flow diagrams, which are used in the functional model. The object model, is an extension of entity-relationship diagram conventions incorporating class operations (methods) and inheritance (in the Smalltalk sense). If you are familiar with these three basic techniques, the OMT methodology shows how information from the dynamic and functional models can gradually be pushed into the object model. OMT provides an evolutionary approach to ease people into the world of OO analysis and design, using existing modeling paradigms. I suppose that I should also mention that the pretty pictures are diagramming conventions that you will already know if you are familiar with the above structured techniques. No three-dimensional dodecahedrons, no dithered lines, no trisected equilateral triangles, etc.

The strengths of the book and the methodology are many. The methodology draws on knowledge of familiar modeling techniques. It is soft and can be tailored in a number of ways for introduction into DP shops currently using structured techniques. The examples presented in the text are excellent since they have been drawn from real-world problems encountered by the authors during the course of their research. Within the context of some examples, the authors describe how subsequent requirements information caused them to go back and adjust their models. They give the reader a view of the model over the life cycle of analysis and design rather than just presenting the "answer." There is very good coverage of some of the design issues involved when trying to incorporate an OO design into systems containing components built with more traditional technologies, such as relational databases. The authors also attempt to provide practical advice about implementing your OO design in non-OO programming languages.

Another strength is that the book can easily be used for reference purposes. Each chapter contains a very thorough bibliography. The organization of the book is such that the reader can focus very quickly on the chapter that is relevant to his or her question. It contains a glossary. The book can be used as a supplemental educational text since each chapter is followed by exercises, with selected answers in the back. Finally, the text is easy to read, which helps if the only time you have for technical books is after your spouse and kids have gone to bed!

And, should your MIS/DP customer wish to compare OMT with other methodologies before going out to buy the latest, greatest CASE tools or white boards, the authors have conveniently included a chapter to make the decision easier. They compare OMT with SA/SD, Jackson structured development (JSD), and conventional ER modeling, describing under what circumstances they believe each model excels.

There are few negative aspects about this book. The methodology may be confusing for people coming from an object-oriented background. The notion of having to map dynamic and functional behavior into methods will be foreign since it is natural for them to think in terms of methods from the analysis stage. For OO types, the object model should be sufficient for the analysis. The chapter on system design is the weakest link in the life cycle chain, but it's also the hardest in real life so, although it does not provide the system design cookbook, it does allude to many of the real-world decisions that are made during this stage of the model refinement. I was

continued on page 18 ...

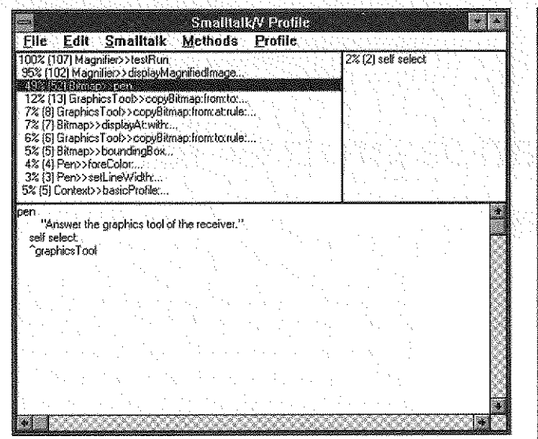


Figure 1. Initial profile.

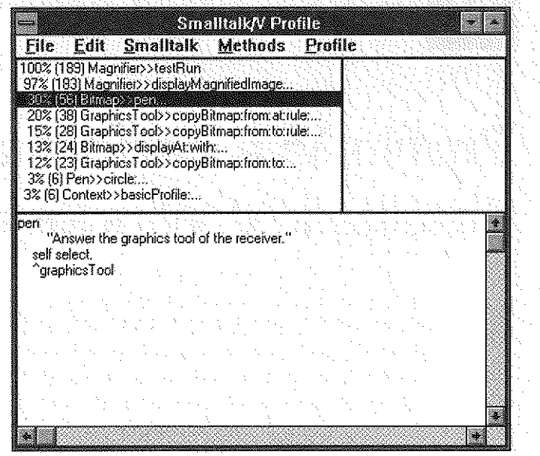


Figure 3. Final profile, with boundingBox message removed.

After making this modification, I again profiled the method, getting the results shown in Figure 2. As you can see, the pen message frequency had been reduced to 23%, which is half of the first run.

And, as shown in Figure 3, you can see that the pen message has increased to 30% of the running time, but the boundingBox message has disappeared, and, as a result, the method runs faster.

So you can probably see by now that this tool is a valuable one. I would never have guessed that the pen message is one to avoid, and, in a real-world application, things like that can mean the difference between acceptable and poor performance.

PROBLEMS WITH PROFILE/V

So far, the only problems I have had with Profile/V are small ones relating to the user interface. One is that the indent on the profile tree is hard to make out since each successive indent is only one space. I spoke with Kent Beck, the author, and he assured me that this had been changed in future versions to make it more readable.

The other problem is perhaps more important and it involves the way the children of a method are hidden and shown. In Profile/V, some of the direct children of a method may be visible, while others are not. This presents problems when trying to view your profile from a given depth since you often have to either do two double-clicks to get the desired results or use the Hide Children menu command. You can get around this by adjusting the threshold to be one (so it only takes one double-click), but personally I think it would be more useful to have a feature that allows the user to set a depth threshold rather than (or in addition to) a percentage threshold.

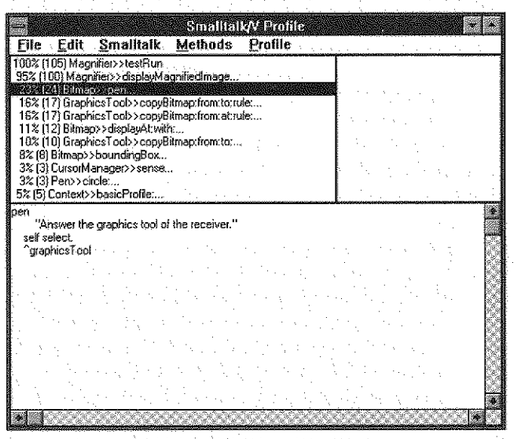


Figure 2. Profile with cached pens.

FINAL WORD

I found Profile/V to be an extremely useful piece of software and I will definitely use it in the future. In comparison, I have only briefly seen the profiler that Digitalk is shipping with Smalltalk/V PM 1.3. It is lacking in that it only produces fairly complex text reports and has no user interface to allow browsing of a profile.

I recommend Profile/V as a solid addition to any serious Smalltalk developer's toolkit. ■

REFERENCES

[1] LaLonde, W. R., and J. R. Pugh. Graphics through the looking glass, *Journal of Object-Oriented Programming*, 1(3), 1988, pp. 52-58.

Jon Hylands is a member of the technical staff at The Object People in Ottawa, Ontario. He is also a part-time student in the School of Computer Science at Carleton University. He can be reached at (613) 230-6897.

observes, being clear about who owns what, or more precisely who is responsible for what, actually goes a long way in resolving conflicts before they get started.

FLEXIBLE PROGRAMMING ENVIRONMENT

I agree with Juanita that "flexibility in programming environments is critical." I disagree with her statement, "Systems with class ownership are not flexible." A good programming environment should be able to maintain flexibility without compromising the integrity and reliability of the classes. The programming environment should be flexible enough to cater to widely different organizational cultures and software environments. It should be appealing to the "rape and paste" rapid prototyper as well as the person who is engaged in production software engineering. In addition, it should be forgiving of the user's mistakes.

In a production software environment, it is often necessary to maintain comprehensive change control over the various software elements; otherwise, system integration becomes a nightmare. In certain organizations, it may be mandated that third party reusable classes not be tampered with, for fear of compromising the integrity and reliability of client code that is dependent on them. Indeed, the reusable class vendor (an internal organization or an outside source) may have shipped a class library without any source. This is eminently possible when classes are packaged as dynamic link libraries. Under these circumstances, even though you cannot modify an existing method, in ENVY you can add extensions to these otherwise read-only classes in your own application.

Juanita notes the difficulty in managing the ramifications induced (vis-à-vis class ownership) by introducing changes in a class hierarchy. She concludes, using an interesting syllogistic argument, that therefore the same developer must own all the classes in the hierarchy. This need not be the case at all. In fact, it is impractical to expect that the superclass and subclass owners be the same. Often times the superclass owner may be a third party vendor or a different organization geographically remote from the subclass developer. In a programming environment such as ENVY with comprehensive version control and configuration management facilities, a complete system consists of a collection of compatible applications. By compatibility I mean, for instance, that the well-being of a subclass client depends upon a properly functioning superclass. Now if the superclass owner makes a change in his class, it may indeed compromise the integrity of the subclass. It is therefore incumbent upon the subclass owner to adapt his class to the newly changed superclass before a new configuration of the integrated system is released. This is no different from the everyday situation where we developers have to port our classes to new versions of the Smalltalk products from vendors.

I agree with Juanita's concluding premise that classes developed by multiple programmers are understood by multiple programmers. I disagree with her observation that class ownership is an obstacle to accomplishing that. Classes in Smalltalk often reflect the style and personality of the author. Having too many developers on a single reusable class may introduce conflicting styles, idioms, and figures of speech that together strike a discordant note to the hapless client. As a flexible programming environment, ENVY recognizes the need for new extensions to existing classes and therefore permits the distribution of protocol among several applications possibly authored by different programmers for ever-so-specialized reasons. The primary author serves as a focal point for the evolution of the reusable class. A class, in the course of its lifetime, may see its author pass on to a different project or even leave the company. Or, the author may want someone else to assume the class' maintenance. Flexible programming environments provide mechanisms for effecting a smooth change of guard to establish a new class owner.

CONCLUSION

The features and philosophy of class ownership (and indeed that of software component ownership) foster a disciplined software environment without compromising the classical productivity gains of Smalltalk. Class ownership itself is inadequate. The ownership mantle has to be pervasively applied across all the different units of software that together comprise a complete system. This requires a programming environment that uniformly applies the ownership philosophy across the various development tools. It should be flexible enough to accommodate different organizational work cultures vis-à-vis team programming.

Class ownership provides a framework for properly separating the activities of component building from application building. Component builders are those people whose major goal is to build reusable components and who should have a reward structure to match. Application builders are trying to get an end user system out the door, and programming for reuse may not be a critical factor for them. Even if developers have to play both roles, it is important that they understand and record the role that they are playing at anytime. Ownership and responsibility for software is a key factor in long-term software quality and reusability. ■

S. Sridhar is a senior member of the technical staff at Knowledge Systems Corp. in Cary, NC where he is actively applying Smalltalk to a variety of software engineering problems. He has also developed substantial applications designed to meet specific customer requirements. He came to KSC from Mentor Graphics Corp. where he was the project lead for Mentor's next generation design management environment developed in C++. Prior to that he worked at Tektronix for four years on Common Lisp and Smalltalk/80 product development. While at Tektronix, he developed numerous tools and components running in the Smalltalk/80 environment. He was an early developer of a framework for delivering stand-alone Smalltalk applications.

Rebecca Wirfs-Brock

Determining object roles and responsibilities

Donald Norman,[1] in *The Design of Everyday Things*, makes the following statement:

> Consider the objects—books, radios, kitchen appliances, office machines, and light switches—that make up our everyday lives. Well-designed objects are easy to interpret and understand. They contain visible clues to their operation. Poorly designed objects can be difficult and frustrating to use. They provide no clues—or sometimes false clues. They trap the user and thwart the normal process of interpretation and understanding. Alas, poor design predominates. The result is a world filled with frustration, with objects that cannot be understood, with devices that lead to error.

I never thought I'd say this, but software objects *are* like real-world objects! Both kinds of objects are hard to use if they are poorly designed. Ensuring that software objects are easy to use involves paying attention to a number of sound design principles. No one ever said that good object-oriented design is easy. In this month's column, I'll discuss the importance of understanding and modeling object roles. Once there is a clear sense of an object's intended purpose, it is much easier to detail the necessary behavior in an understandable fashion.

Identifying the central classes in an application is just the first step. Combing through a specification of the problem may provide an initial list of candidate classes, but what next? First, let me state that no designer I know has ever found all the key objects by reading and understanding a specification of the problem. A specification is just a launch pad for design activity. Depending on the weight of that specification, there will be different strategies needed to find those key classes. If there is a mound of paper to wade through, the initial task will be one of filtering out a lot of detail and focusing on identifying the highest level concepts. On the other hand, if the specification is on the slim side, the task will be to develop a skinny statement of intent into a model of key concepts that will drive the design.

There is a deceptively simple question that needs to be answered for each identified class. Can that class' purpose within the application be clearly stated? I've found it useful to force myself to write a concise, precise statement of purpose for each potential class. This purpose statement need not be long or wordy; a sentence or two will often suffice. However, if it is difficult to construct a succinct statement, more work is needed. There are several plausible explanations (other than that the class doesn't belong in the design) for being unable to write a clear purpose statement for a class.

SUBDIVIDING LARGE CONCEPTS

For one thing, the class may represent too large a concept. One indicator of this is that the class seems to embody an entire program or a major portion of the overall system behavior. This large concept needs to be decomposed into more understandable pieces. What are the constituent responsibilities of this mega-object? To answer this question, we must resolve a rather complex concept into simpler, more basic ones. These simpler concepts will be easier to understand, and their purpose and role will be easier to elaborate. Simpler concepts will be represented by classes in the final design, while the larger concept may not.

> “ ... software objects **are** like real-world objects ”

It is conceivable that the large, vague concept still has a role to play and will be represented by a class in the final design. For example, the object might be responsible for coordinating the actions of other objects (each with a concisely stated purpose) that collaborate to fulfill the larger purpose. One design for an automated teller machine might have an automated teller session object whose purpose is to conduct a customer session. This customer session would consist of a series of user transactions with the bank (and a whole chain of responses to user requests) that are coordinated by the automated teller object.

Subdividing the responsibilities of a large, complex class into a number of simpler classes requires deeper understanding of the system. Each newly created class needs a clearly stated

Reviewed by Jon Hylands

Profile/V: a performance profiler for Smalltalk/V Windows

Profile/V, from First Class Software, is a code profiling tool that allows Smalltalk programmers to monitor the performance of their applications. It creates a weighted call tree of your code that basically shows the percentage of total running time spent in each method. With this information, it is possible to find out where your code (or, just as important, system code) is causing a bottleneck.

With a list price of $299.99, Profile/V is a tool that any Smalltalk programmer who is interested in writing high-performance code should include in their library. Although it needs some improvement in the user interface department, it is definitely money well spent. It is currently available for Digitalk's V Windows,V Mac, and V 286. Profile/V will be available for V PM this month.

HOW TO USE PROFILE/V

Profile/V comes on one software diskette and includes a 50-page *User's Guide/Tutorial*. The manual's 29-page tutorial shows the optimization of a simple graphical application, which is included on the disk. The manual also includes sections on installation, how to use the product, notes on how it is implemented, and a very interesting section on "Programming for Optimization."

The only problem I had with the manual is the fact that the installation page is somewhere in the last half—when I look for the installation instructions, I expect them to be at the beginning.

Profile/V uses an invisible window to capture timer events and takes a snapshot of the stack from the current user interface process when a timer event happens. It builds a profile object from these samples and then can open a browser on the profile. The browser is a subclass of the system-supplied method browser. The browser has three panes and it provides the user with the ability to go as deep as they want—right down to individual statements in a method.

Other valuable features include the capability to gather method profiles for the same method and browse them as a new profile. This feature is ideal when profiling recursive methods. Another useful utility is the ability to take what is displayed in the browser and convert it into formatted text in a workspace for inclusion in documents (such as this one). You can also adjust the threshold value for the browser, which controls how many methods are shown when the browser is initially opened by hiding all methods that take less than the threshold percentage value to run.

Perhaps one of the nicer things about Profile/V is its size, or lack thereof. The entire profiling system is only about 27K of source code, which makes it a product more likely to be understandable and extendable.

BUT MY CODE IS ALREADY FAST...

Many programmers, myself included, will look at this tool initially and say something to that effect. Unfortunately, in the case of Smalltalk, where you have a large library of reusable code *written by someone else*, having your code run at light-speed doesn't necessarily mean your application will be as fast as it can be. Programmers tend to make assumptions about the performance of other code, and these assumptions often turn out to be incorrect. This turned out to be the case for a graphics application I profiled.

USING PROFILER TO OPTIMIZE A SAMPLE APPLICATION

The application I ran my tests on was a simple magnifying glass, which first appeared in the Smalltalk column in the *Journal of Object-Oriented Programming*.[1] Since that time, the authors have made large number of changes to the code to simplify and streamline it. The magnifier simply simulates a magnifying glass on the screen and shows the magnification of a circular area. I limited the tests to a single method, which is the code that displays this circular magnified image, since it is the slowest part of the magnifier simulation.

The first iteration of the profiler run on this method produced the profile shown in Figure 1. It shows quite clearly (and quite surprisingly, also) that almost half the time spent in this method is in sending the **pen** message to bitmaps!

The pen message is sent six times since we are performing five copyBitmap's and one set of drawing commands to achieve the circular magnification effect. However, we can improve this since only two bitmaps are the receivers of the pen message. We can cache each bitmap's pen in a temporary variable at the beginning of the method, thus saving four pen messages. This works when performing copyBitmaps, but not when doing pen-based drawing, so the pen message must also be sent before the drawing section of the method takes place.

where the method max: is located in class Number. Then in the following:

```
| (temp:Integer) (index1:Integer) index2 |
temp := index1 max: index2.
```

the message max: would be bound to the method max: in class Number at compile time, and an instance of BoundMethod would be entered in the global Set, BoundMethods. This instance of BoundMethod would contain a BehaviorConstraint. The compiler rule used to determine whether a BehaviorConstraint or a TypeConstraint is generated is fairly simple. If a method is redefined in any of the subclasses of the constraint class, the compiler will generate a TypeConstraint. If such redefinition does not occur, the compiler will generate a BehaviorConstraint.

If the method max: was now defined in class Integer, the presence of a BehaviorConstraint in BoundMethods would inform us that there was a "sending" method that required recompiling, and BoundMethods would be updated to reflect the new BehaviorConstraint.

If the method max: were now defined in class SmallInteger, the compiler (using the rule mentioned above) would remove the BehaviorConstraint and substitute a TypeConstraint. In our system, BoundMethods must be loaded at system start-up since they will be invoked by direct function call.

Dynamic binding would remain the primary and preferred way of associating messages with methods. Typing would be used in situations that caused performance degradation or as a data validation tool. Intuitively, the best use of typing applies in high-use areas where typed languages can typically produce very efficient code. Coincidentally, these areas correspond to functions in Smalltalk that undergo few changes since they are integral to the basic functioning of the system. Some example preliminary candidates for typing might be arrays, which are frequently used in the at: and at:put: messages, and array indices, which participate in integer operations. In some actual program samples we have studied, up to 40% of message routing would be removed by static binding in these areas.

Typing will probably be a compiler option that may be turned on or off by the programmer. Programs compiled for production would usually take the performance advantage of typing, while, in the development environment, typing might not be used to retain flexibility and fast compilation. ■

Listing 1.

```
Association subclass: #ConstrainedAssociation
    instanceVariableNames:
        'constraint '
    classVariableNames: ''
    poolDictionaries: ''

ConstrainedAssociation class methods

key: aKey value: anObject constraint: aClass
    "Answer an instance of class ConstrainedAssociation
    whose key is initialized to aKey, whose value is initialized
    to anObject, and whose constraint is initialized to aClass."
    aClass isBehavior
        ifFalse: [ ^self error: 'constraint must a Class '].
    (anObject isKindOf: aClass)
        ifFalse: [ ^self error: 'value must be kindOf', aClass name ].
    ^( (self key: aKey) value: anObject ) constraint: aClass

ConstrainedAssociation methods

constraint: aClass
    "Set the constraint of the receiver to be aClass. Answer the
receiver."
    aClass isNil
        ifFalse: [
            (value isKindOf: aClass)
                ifFalse: [
                    ^self error: 'value must be kindOf ', aClass name ] ].
    constraint := aClass!

value: anObject
    "Set the value of the receiver to be anObject if anObject
    is an instance of constraint or one of its subclasses."
    constraint isNil
        ifFalse: [
            (anObject isKindOf: constraint)
                ifFalse: [
                    ^self error: 'value must be kindOf ',
                        constraint name ] ].
    value := anObject
```

REFERENCES

[1] Johnson, R. E., J. O. Graver, and L. W. Zurawski. TS: an optimizing compiler Smalltalk, *OOPSLA '88 Conference Proceedings*, San Diego, CA, October 1988, pp.18-26.

[2] Chambers, C., and D. Ungar. Making pure object-oriented languages practical, *OOPSLA '91 Conference Proceedings*, Phoenix, AZ, October 1991, pp. 1-15.

[3] Palsberg, J., and M. I. Schwartzbach. Object-oriented type inference, *OOPSLA '91 Conference Proceedings*, Phoenix, AZ, October 1991, pp. 146-161

Glenn J. Reid is President and Founder of QSYS Systems Consultants, Inc., a consulting and software development company whose main area of expertise is in the application of object-oriented technology. Architect of Smalltalk/370, Mr. Reid is currently involved in the development and application of a complete project life cycle approach to developing object-oriented systems in a mainframe environment. He can be reached at (416) 343-6464.

role. There already may be identified classes that can fulfill part of the responsibilities of the rather large concept. Most likely, this isn't the case. A hypothesis must then be formulated on how to partition the vague concept into several distinct roles. Each role will be assigned to a new class. A key designer of a large, successful application told me that his design team subdivided responsibilities according to when, what, and how. These subresponsibilities were then assigned to separate classes that were either responsible for knowing when, knowing what, or knowing how to perform an operation. Sounds simple enough. The design team found they spent time debating whether a particular responsibility was actually a when, a what, or a how. One object's what is another object's how. It all depends on a particular point of view. At least the team had a strategy for elaborating class roles. But they still had to debate the details in context of their emerging model.

COMPLETING A MODEL OF OBJECT INTERACTIONS

There are other situations where it is difficult to state a class' purpose. One common situation is that a class doesn't seem to be connected to any others. It's hard to explain why this disjoint class should exist, yet the designer remains convinced that it's important. Chances are, the class is important. The problem is that the model is incomplete. This problem typically arises when classes are sifted through one at a time, rather than building an understanding of the collaborative behavior between objects in the design.

To understand any single object's role, it must be looked at in the context of others with which it interacts. Constructing an object-oriented design is not a linear, top-down process, although it is often to present the design that way. Understanding an object's purpose forces the designer to understand the roles of other objects. To understand the role of a seemingly isolated object, both an understanding of its static, structural relationships with other objects and interactions with other objects is needed.

To determine the static relationships an object has with others, examine how an object is connected to others. Is there a whole–part relationship between it and another object? Does this object represent an aggregation of other objects? If so, it is usually pretty simple to fit this object into the design.

It is much harder when an object participates in a number of relationships. In this case, it is useful to build an understanding of the dynamic behavior of the object. Performing design walk-throughs by tracing a chain of object collaborations in response to a stimulus is a good way to understand object interactions. Ivar Jacobson,[2] pioneer of the Objectory method, introduced the notion of *usage cases*. Usage cases can be recorded and then used to test the model under both normal and abnormal conditions. A key component of Steve Weiss and Meilir Page-Jone's[3] object-oriented software synthesis method is modeling the response to events and understanding their impacts on a design. The idea behind both techniques is to translate requirements into events and to associate events with objects that are responsible for handling them.

The more situations that are modeled, the better. As simple as this sounds, it takes some skill to effectively elaborate object interactions. The goals is to first develop a "big picture" before diving into detail. The way to do this is to trace object collaborations between objects that are at either the same or next conceptual level in the design. First, develop an overall, high-level view of key object interactions. Then elaborate and subdivide roles and object responsibilities. This breadth-first approach avoids modeling classes at widely differing conceptual levels, which indeed is difficult.

This breadth-first approach represents an ideal. In practice, some areas of the design will be better understood and naturally elaborated before others. An uneven design model can make it difficult to trace object collaborations. It will be relatively easy to trace the collaborative behavior throughout the well-understood parts of the design. When collaborations are necessary with objects in an undeveloped area, suddenly what had seemed straightforward becomes very unclear. This isn't a sign of failure; it just indicates that the unclear part needs elaboration.

OBJECTS THAT DON'T FIT THE MODEL

Perhaps one of the toughest problems to deal with is when an object doesn't fit with the designer's notion of what constitutes a "good" object. It is very difficult to explain the purpose of such misfits. Criticisms commonly leveled against such troublesome objects are:

- This is an organizing object. It is too simple. It merely consists of data. It has no behavior. Aren't objects supposed to have both?
- This object's only purpose seems to be to route messages between two other objects. Why should I have intermediary between these objects? Can't they just directly communicate with each other instead?
- This object is too action oriented. Aren't objects supposed to encapsulate both operations and data? This object seems like a pure "process." We're doing an object-oriented design, not a process decomposition.

> Constructing an object-oriented design is not a linear, top-down process, although it is often useful to present the design that way.

There are no pat answers to these criticisms. In each case, the object doesn't match the designer's expectations. The model, the designer's expectations, or both need readjustment. In the first case, it is worth noting that objects are not uniform packages of operations and data. It is natural that the proportion of each will vary according to the object's role in the design. It is perfectly reasonable for relatively simple objects to coexist alongside more complex ones. However, the object must stand on its own merit to be included. Indeed, there may be preferable alternatives to creating a "data mostly" object.

When creating an object model, the designer may need to invent mechanisms that weren't spelled out in the specification. Mechanisms may be added for the express purpose of reorganizing the flow of information and communication between objects. These mechanisms may help reduce object coupling or provide an abstract connection between objects. The consequences of inserting such mechanisms needs careful consideration. But, objects whose purpose is to organize or manage communication between objects can be reasonable design additions.

In the third case, the purpose of an object may be to transform information from one form to another. Such process-oriented objects can naturally occur in a design and are not always a sign that the designer hasn't shifted from the procedural to the object-oriented paradigm. Each process-oriented object should apply a fair amount of intelligence to produce results. Better yet, a process-oriented object can often provide a completely different view on the transformed information. The objects being processed and the clients requesting the transformed information may be only dimly aware of each other. In this case, the process-oriented object is probably a reasonable design concept. One example of a process-oriented object is a compiler. The role of a compiler is to transform text into an executable program structure. It takes a lot of intelligence to perform this operation. Defining a compiler object is a reasonable design choice.

It may be that a class doesn't belong in the final design. *Websters Dictionary* defines role as "a character assigned or assumed. A part played by an actor or singer." The task of the designer is to assign each object an appropriate role. Each role is constrained to fit within the existing object model, but a lot of designer discretion is still involved. It's a challenge to design well-understood, easy-to-use objects. But the positive impacts that well-designed objects have on application maintenance and understandability are well worth the extra effort. ■

REFERENCES

[1] Norman, D. *The Design of Everyday Things*, Bantam-Doubleday-Dell, New York, 1988.

[2] Jacobson, I. Object-oriented development in an industrial environment, OOPSLA '87 Conference Proceedings, Orlando, FL, *SIGPLAN Notices*, 22(12), 1987, pp. 183-191.

[3] Weiss, S., and M. Page-Jones. Synthesis/analysis and synthesis/design, *Proceedings of the Object-Oriented Systems Symposium*, Summer 1990.

Rebecca Wirfs-Brock is the Director of Object Technology Services at Instantiations and coauthor of Designing Object-Oriented Software. *She is the program chair for OOPSLA '92. She has sixteen years of experience designing, implementing, and managing software products. During the last seven years, she has focused on object-oriented software. She managed the development of Tektronix Color Smalltalk and has been immersed in developing, teaching, and lecturing on object-oriented software.*

gram. The program is typeable if these constraints are solvable. Static binding information is derived from the solution. This project is currently implementing the inferencing algorithm, with an optimizing compiler as a future undertaking, and so has no performance results to report. In our initial exploration of explicit typing within Smalltalk, we have confined ourselves at this time to investigating typing of named variables, excluding such things as intermediate results generated during expression evaluation. Potential candidates for typing are:

- dictionary variables (i.e., class, pool, and global variables)
- instance variables
- arguments
- named temporaries
- receivers

In all cases, the affected variable would be constrained to belong to a particular class or one of its subclasses (i.e., the variable has been "typed"). Thus, we are using a simpler version of typing than that used in Typed Smalltalk. For this discussion, type and class may be considered synonymous. Here are some of the issues involved.

For programmer convenience, we would prefer a common type declaration syntax that could be used for all the above-mentioned cases. A possible candidate syntax is shown below:

```
Current Smalltalk:   variableName
Typed Smalltalk:     ( variableName:Class )
```

This new syntax would be used wherever variables are "declared" in Smalltalk, that is, in class definitions, message patterns, and declarations of temporaries.

Typed variables must be initialized according to type. Untyped variables are initialized at creation with the value nil. This is unacceptable for typed variables. If variable `x` is declared as:

```
( x:Array )
```

we must ensure that `x` always contains an `Array` object; otherwise, invocation of the statically bound expression:

```
x at: 1
```

would have disastrous results. This requires a modification to the new and new: methods of class `Behavior`. Variables typed as `Data Types` (i.e., types that do not have a direct system representation of their data structure) would be initialized by sending the message new to the appropriate class. Variables typed as basic `Data Structures`, such as `Integer`, `Float`, and `Array`, would be initialized at the primitive level. A possible set of initialization values for `Data Structures` might be:

```
Integer 0
Float   0
Array   0 elements
```

This arrangement would cover most cases, including the special initialization requirements that apply to some classes (e.g., `OrderedCollection`). Immutable `Data Types` (e.g., `Character`) that disallow creation of new instances present some difficulties, the main being that it is currently impossible for the system to determine whether a class is immutable.

It would possible to initialize typed temporary variables in methods to nil, as is done currently, since the compiler would recognize that these variables remain untyped until an assignment takes place, at which point the typing could then be taken into account. This might be preferable to reduce initialization overhead.

Runtime type checking is required to ensure that typed variables are assigned according to their declared type. This function would be performed by compiler-generated code that would perform the equivalent of an `isKindOf:` check prior to assignment of an expression result. The system overhead of this check is minimal. In addition, typed arguments would be checked upon entry to a method.

At compile time, Smalltalk changes a reference to a `Dictionary` variable into a reference to the `Association` containing the variable key and value to avoid a runtime dictionary lookup. However, dictionary variables may be updated through basic `Dictionary` messages `at:put:`, `removeKey:`, etc. This creates the potential for an integrity violation in Smalltalk (try removing an existing class variable with `removeKey:` then adding it again with `at:put:`). While it could be argued that one should not update dictionary variables in this manner, nevertheless it is an option open to the user. This situation is aggravated for typed `Dictionary` variables since there is no compiler-generated code to stand in the way of an incorrect assignment when using the basic `Dictionary` messages. Our present solution to this is to create a subclass of `Association`, call it `ConstrainedAssociation`, that would contain a new instance variable, `constraint`, and would inhibit incorrect assignment to its value. The class definition and methods for `ConstrainedAssociation` are shown in Listing 1. Note that our solution does not address the `removeKey:` integrity problem that currently exists in Smalltalk.

To manage static binding, we propose creation of the classes:

```
BoundMethod
Constraint ( virtual class - no instances )
    BehaviorConstraint
        TypeConstraint
```

`BoundMethod` would be a tuple containing at least an "implementing" `CompiledMethod`, a "sending" `CompiledMethod`, and an instance of either `BehaviorConstraint` or `TypeConstraint`. `BehaviorConstraint` describes an instance of static binding, and `TypeConstraint` describes the less restrictive case of simple type checking.

In the following example, let us assume the class hierarchy:

```
Number
    Integer
        SmallInteger
```

SMALLTALK COMES TO THE MAINFRAME, PART 2

Glenn J. Reid

In part 1 of this article, we discussed our implementation of Smalltalk in an IBM mainframe environment that we have called Smalltalk/370. Mention was made that we are investigating the introduction of typing into Smalltalk, currently a popular area of interest in the OO community. Here, in part 2, we will discuss some specifics of our investigation (not yet complete) and, hopefully, shed some light on the difficulties involved in typing a language like Smalltalk.

Before we launch into a discussion of solutions, perhaps it would be appropriate to determine what we are investigating and why. In part 1, we stated that the performance overhead of dynamic (or late) binding would probably be unacceptable in Smalltalk/370, particularly since degradation of the system affects all users in a time-sharing environment. The fastest untyped version of Smalltalk today is the ParcPlace Smalltalk-80 implementation, which runs at approximately 10% the speed of optimized C. This does not imply that the basic mechanism of dynamic binding in Smalltalk must be thrown out. As with many performance problems, it is very possible that concentrating on a few areas of concern will lead to a satisfactory system. Since dynamic binding is the problem, we must substitute statically bound procedure calls or, better yet, in-lined procedures in the areas where they provide the most benefit.

When we first considered the dynamic binding problem, we felt that we would probably be able to implement a static typing mechanism that would allow programmers to explicitly declare variable types within their programs and enable our compiler to make use of these for optimization purposes. In our first attempt, we limited the scope of our ability to explicitly type Smalltalk. Since performance was our main goal, rather than a comprehensive typing system, we considered this approach acceptable. We have included a few of the details of this approach later in this article.

As our static typing mechanism gained in substance, a number of things became apparent. Explicit type declarations increase system complexity from the users perspective. A new dimension is required in programmer thinking. Not only must performance requirements be observed in producing algorithms that operate efficiently, but all variations of types that the algorithm could operate upon must be considered as well as the relative volume of message sends to each type. It is possible that subsequent changes to the system may make previous tuning invalid. Furthermore, type declarations may restrict the application of a method. For example, a method argument may be typed, causing method failure, similar to primitive failure, when an argument with an incorrect type is received. This makes the programming environment less flexible, or more complex from the programmers point of view. Intuitively, it appears that these complexities will increase if we expand our typing strategy.

As part of our investigation, we are reviewing published literature in the area of typing and optimizations to pure object-oriented languages. So far, we have come across three different approaches.

Probably the furthest advanced example of comprehensive explicit typing within Smalltalk is the Typed Smalltalk (TS) project.[1] In this project, a syntax extension to Smalltalk allows the programmer to explicitly declare types for variables, method results, etc., that the compiler can use to statically bind or in-line procedures. Published performance results indicate that some small benchmarks have achieved speeds at least twice that of Smalltalk-80. Since this data is not recent, and we understand that work is still continuing on Typed Smalltalk, we expect that these results have been improved still further. This approach is closest to our initial experiments with explicit typing. Since this project is much further advanced, we will probably look to it to evaluate some of our concerns mentioned above.

Recent benchmark results in the SELF programming environment have demonstrated a Smalltalk-like language running at approximately 57% of the speed of optimized C.[2] These results were achieved without the introduction of explicit typing within the language. In this approach, the compiler uses "path splitting" to generate both high- and low-performance paths through a method. Path splitting is used when a frequently used message selector whose receiver usually belongs to a particular class is detected within the source program. For example, path splitting would occur for the high-frequency message at:, which is most often received by an instance of Array. This approach uses the advanced techniques of dynamic compilation, customization, deferred compilation, and path splitting management algorithms to produce the results mentioned above.

Finally, we have noted that some are working in the area of type inference without explicit typing.[3] Here, a type inferencing algorithm constructs a graph of type constraints from a pro-

GETTING REAL

Juanita Ewing

How to use class variables and class instance variables, part 1

In last month's column, I discussed some strategies for initializing classes and how initialization related to class variables and class instance variables. In this column, I will talk about coding conventions for class variables and when to use class variables vs. class instance variables.

Classes that use class variables can be made more reusable with a few coding conventions. These coding conventions make it easier to create subclasses. Sometimes developers use class variables inappropriately. Inappropriate use of class variables results in classes that are difficult to subclass. Often, the better implementation choice for a particular problem is a class instance variable instead of a class variable.

WHAT ARE CLASS VARIABLES?

Classes can have:

- class variables
- class instances variables

Class variables are referenced from instance and class methods by referring to the name of the class variable. Any method, either a class method or an instance method can reference a class variable. Figure 1 contains a diagram of a class, ListInterface, that defines a class variables.

The methods in ListInterface would look like this:

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

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

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

        ^ListMenu notNil

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

        self hasMenu
            ifTrue: [^ListMenu startUp].
```

Both instance and class methods can directly reference class variables by name. The class method initialize is used to bind values to the class variables. The instance methods hasMenu and performMenuActivity reference the class variable ListMenu. All instances of ListInterface and the class ListInterface share the same class variables.

HOW ARE CLASS VARIABLES INHERITED?

Class variables and the values they are bound to are inherited. The class variable referenced by a subclass is the same as the one referenced by the superclass. This means that a class variable is shared by a class, all its subclasses, and all the instances of the class and its subclasses.

> "It is possible for subclass methods to modify inherited class variables, but generally it is undesirable to do so."

Our example has a subclass of ListInterface called CalculatedListInterface. Subclass methods referring to the ListMenu class variable reference exactly the same object as the superclass method. The subclass CalculatedListInterface has behavior that is different from its superclass, as defined by the method conditionalMenuActivity:

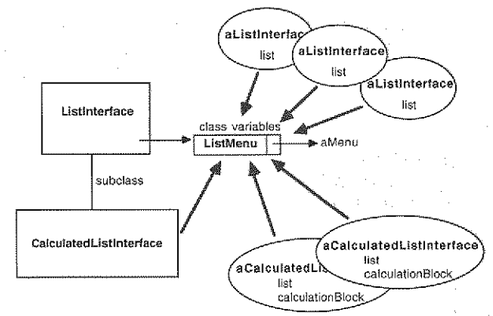


Figure 1. Class variables are refernced by subclasses and all instances.

```
CalculatedListInterface
    conditionalMenuActivity
        "Perform the mouse-based activity for my view if the list is not
        empty. If there is no menu, flash the list pane."

        self hasMenu
            ifFalse: [^self flash].
        list isEmpty
            ifFalse: [^ListMenu startUp].
```

Subclass methods can directly reference class variables that are defined by the superclass. In our example, the CalculatedListInterface method references the class variable ListMenu that is defined by ListInterface. This is different from the inheritance of instance variables. The method conditionalMenuActivity references the instance variable list that is defined by the class ListInterface. But, each instance of CalculatedListInterface and ListInterface has its own copy of list and does not share its instance variables.

HOW DO SUBCLASSES MODIFY CLASS VARIABLES?

It is possible for subclass methods to modify inherited class variables, but generally it is undesirable to do so. If a subclass were to modify a class variable, it would change the only existing value of the class variable. Each subclass does not have its own copy. It references a shared copy. Generally, developers want to create a new class variable and use it in place of the inherited class variable.

Using our example, we will create a new menu in the subclass CalculatedListInterface. The menu is implemented with a class variable so it is not possible to change the menu for the subclass without also changing it for the superclass. This is because both classes reference the same variable.

The only way to create a new menu for the subclass and retain the original menu for the superclass is to create a new class variable. In our example, we call the new class variable CalculatedListMenu. In addition to a new class variable, all methods that reference the original menu must be overridden in the subclass:

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

        CalculatedMenu := Menu labels: #('add' 'remove' 'print')

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

        ^CalculatedMenu notNil

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

        self hasMenu
            ifTrue: [^CalculatedMenu startUp].
```

Because direct references to the class variable ListMenu are sprinkled throughout the class ListInterface, the subclass must override many methods. In this simple example, we had to override three methods that reference ListMenu to reference a different menu. In a complicated real-world application, many other methods may need to be overridden to reference a different class variable in a subclass. Because significant portions of the class needed to be overridden, the class is not very reusable.

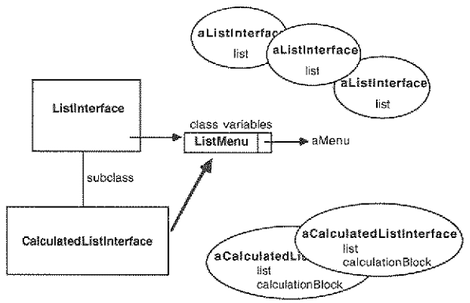


Figure 2. Coding conventions increase the reusability of classes implemented with class variables.

A better version of ListInterface has the minimum number of references to a class variable—one for setting and one for retrieving the value of a class variable:

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

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

    menu
        "Return the list menu."

        ^ListMenu

ListInterface
    hasMenu
        "Return true if a menu is defined."
        ^self class menu notNil

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

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

> Because of the nature of the data stored in class variables, it is best for class methods to store and retrieve the class variables.

This coding convention reduces the number of direct references to a class variable, as illustrated in Figure 2. It is easier to create subclasses because only the methods that set and retrieve the class variable need to be overridden. Now the code for CalculatedListInterface looks like this:

```
CalculatedListInterface class
    initialize
        "Create a a computed list menu."

        CalculatedMenu := Menu labels: #('add' 'remove' 'print')

    menu
        "Return the list menu."

        ^CalculatedListMenu
```

This coding convention effectively restricts the references to a class variable. Because of the nature of the data stored in class variables, it is best for class methods to store and retrieve the class variables. In effect, we have eliminated the sharing between classes and instances.

By eliminating this sharing, we have made ListInterface more reusable; however, ListInterface still has another problem. Another class variable had to be created by the subclass to provide a different menu. Now CalculatedListInterface has two class variables, one of which (ListMenu) is not used.

The root of the remaining problem is that class variables are shared by a class and its subclasses. In our example (and in many other situations), this sharing is inappropriate. Instead, a subclass needs to be able to override inherited data. Class variables share the data between subclasses and superclasses, so it's not possible for a subclass to override the data. Next month, we will explore another mechanism, class instance variables, that will solve our problem. ☒

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

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:

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.

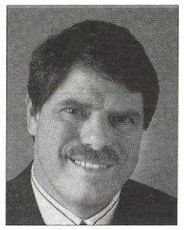
THE PROJECT AND ITS CHRONOLOGY

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

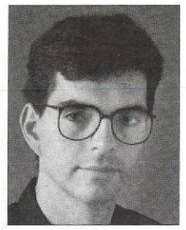
The prototyping stage

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. 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.**

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

```
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 re-engineering 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.

...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...


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

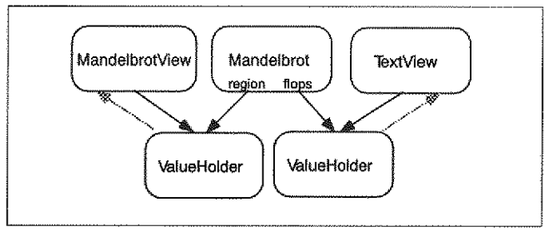


Figure 2. ValueHolder style separation of model and interface.

```
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:

"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 hard-wired 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
```

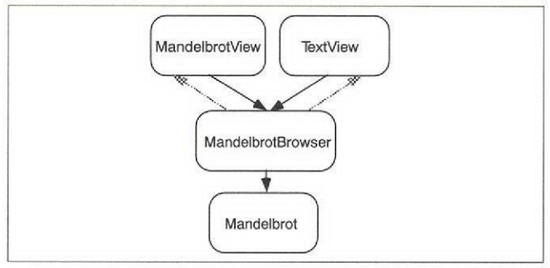


Figure 1. Classic separation of model and interface.

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

SMALLTALK IDIOMS

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

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.

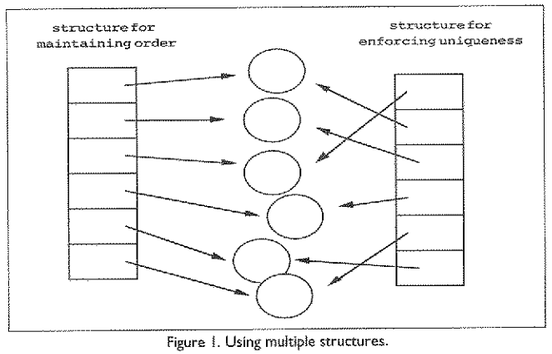


Figure 1. Using multiple structures.

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.

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.

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, SDGraphicGroup 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.

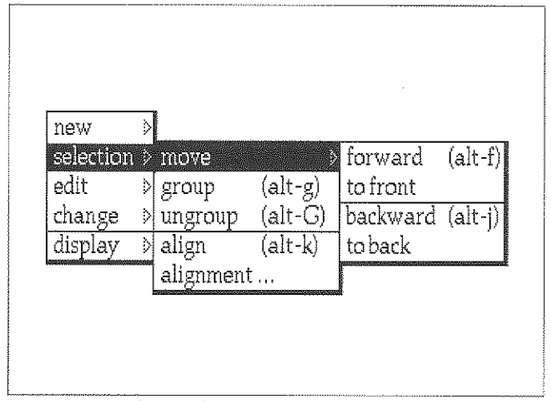


Figure 1. Menu selection for moving objects.

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.

GETTING REAL

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

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...

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.

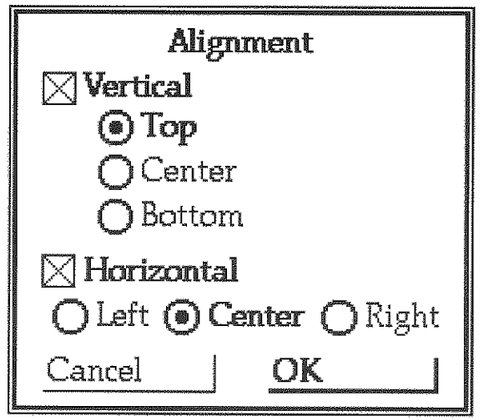


Figure 2. Alignment Dialogview.

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]).
```


> 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

THE BEST OF comp.lang.smalltalk

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-**


```
    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
```

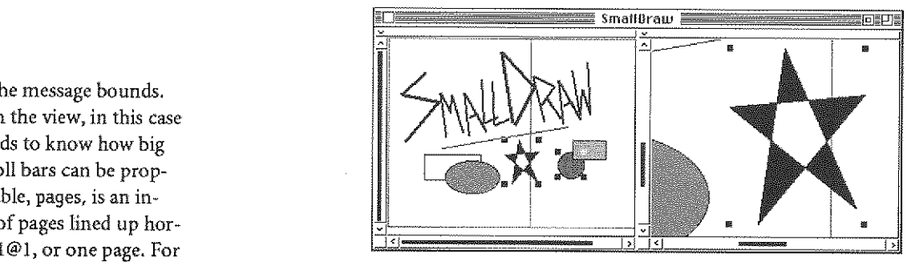


Figure 4. Two scrolling views (25% and 100%) and two pages side by side.

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.

The Smalltalk Report

The International Newsletter for Smalltalk Programmers

November/December 1992 Volume 2 Number 3

TAKING EXCEPTION TO SMALLTALK, PART I

By Bob Hinkle & Ralph E. Johnson

Contents:

Exception handling is an important part of many languages. Although not provided in the original Smalltalk-80 or in Smalltalk/V, it is supported in the latest version of ParcPlace's Smalltalk-80. This article will show how to build an exception handler for any version of Smalltalk and will use Smalltalk/V 286 as an example. Along the way, we'll show you why it's useful for languages to treat seemingly internal mechanisms such as processes and contexts as first-class objects.

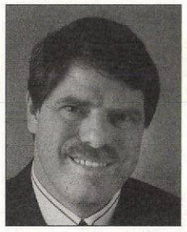
The exception handler was first built for an early version of Tektronix's Smalltalk-80. It was modeled after a version described in an article by Evelyn Van Orden,[1] and we used it in the type inference system of Typed Smalltalk.[2] When we ported Typed Smalltalk to ParcPlace Smalltalk, we wanted to use their faster exception handler, so we modified ours to be compatible. Thus, our exception handler is similar to ParcPlace's, but less powerful. We then developed the V 286 version described here, both to test the generality of the solution and to make the work interesting to a wider audience.

A QUICK LOOK AT EXCEPTIONS

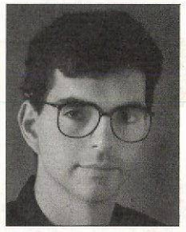
Briefly speaking, exception handling is the provision for non-lexical flow of control in a program when something out of the ordinary (i.e., exceptional) occurs. An exception handler is a part of the program (usually a block in Smalltalk) that can deal with some possible but unlikely event, such as reading past the end of a file, dividing by zero, or referencing out of bounds in an array. In the usual scheme, a program registers an exception handler for a particular kind of event and then continues with its normal processing. If an exceptional event does occur, a signal is raised as a notification to the system. The system finds the last handler that was registered for that signal by searching down the context stack. If one is found, control passes into the exception handler. Depending on the system, the handler will have different options. The handler can usually make whatever changes are necessary; execution can then resume where the signal was raised or where the handler was registered, or return from where the handle was registered.

This description shows that implementing an exception handler requires access to processes and their context stacks. An exception needs to search the context stack to find the correct handler for a given signal and implement non-local control flow. As a result, exception handling could only be added to traditional languages by the language designer. In Smalltalk, however, where processes are objects and contexts can be objects, exception handling can be added by a programmer. Smalltalk's first-class treatment of contexts is one aspect of a concept called *reflection*, which is the idea that languages and systems should objectify their internal mechanisms to make them accessible to the programmer. In that way, programs can monitor and change their behavior, in a sense reflecting on themselves. Our example of exception handling shows how some reflectiveness makes a language more adaptable.

continued on page 3...



John Pugh



Paul White

EDITORS' CORNER

Another OOPSLA has come and gone. This conference represented a significant milestone, both personally (since it's finally done and behind us!) and as Smalltalk users. Based on this conference, it would appear the language wars of the past are now over. Smalltalk is definitely well-entrenched as the language of choice within many organizations and few, if any, of the so-called research-language–type complaints about Smalltalk were to be found. Smalltalk has clearly made it.

Interestingly, the void left by the language wars seems already to have been filled by a full-fledged, drag-em-out war over methodologies. It seemed there were nothing but methodology tools vendors on the exhibit floor. Many were designed specifically for methodologies such as Booch or Rumbaugh, while others were "applicable to all methodologies" (which, of course, more often than not means "useful for none").

Two aspects of this methodology war are worth noting. First, it is not clear that any one will emerge as the winner. That is not such a bad thing. Just as no one language is appropriate for all applications, even within an organization, no one methodology should be applied universally. Like the language wars before it, though, this plea for reason and tolerance will likely be lost among the battle cries.

The second and more subtle aspect of this war is that these methodologies seem better geared for the C++ world. Smalltalk developers seemed, for the most part, removed form the debate. They talked much more about tools that would help you deliver and much less about methodologies. We will have more to say on this subject and the need for better tools that go beyond any particular methodology in future issues.

It is with great pleasure we introduce Ralph Johnson and Bob Hinkle, two well-known members of the Smalltalk community, as our featured writers this month. Over the next few issues, they will address in detail the issue of exception handling using Smalltalk. This is a topic important to all computing languages and one that is often misunderstood. In their opening article, they describe the interface for their exception handler, along with the machine-independent aspects of its implementation.

Also in this issue, Kent Beck continues his survey of the Collection classes, highlighting interesting facts about many of the more popular classes. Rebecca Wirfs-Brock speaks about the need for properly described classes and applications. Juanita Ewing describes a straightforward mechanism for managing source code on small projects. Greg Hendley and Eric Smith survey the events supported by PM's Pane classes. Richard Peskin reviews John Bourne's new textbook, written for engineering programs that introduce the object-oriented paradigm. Finally, Alan Knight returns with more discussion from the USENET world.

Happy holidays to all!

212-274-0640

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 Inc., 588 Broadway, New York, NY 10012 (212)274-0640.

Mailed First Class. Subscription rates 1 year, (9 issues) domestic, $65, Foreign and Canada, $90, Single copy price, $8.00. POSTMASTER: Send address changes and subscription orders to: The Smalltalk Report, Subscriber Services, Dept. SML, P.O. Box 3000, Denville, NJ 07834. Submit articles to the Editors at 91 Second Avenue, Ottawa, Ontario K1S 2H4, Canada.

ples is cursory at best. There is a need in this section for emphasis on real examples. The non–electrical engineering coverage is understandably the weakest, but his circuit simulation example is again too detailed with emphasis on code rather than simulation of physical behaviors. As in prior parts of the book, details of extraneous subjects take up too much space—the external interface description is a notable example. While Bourne does not face some critical issues in engineering applications of Smalltalk, such as handling of large numbers of objects generated in technical computations, he does address performance problems with a discussion of user primitives. However, he confuses user primitives (which are limited by the context loss across calls in PPS release 4) and a true C interface (not yet released for PPS at this writing). Table 12.2 illustrates the serious problem with this book. It is a method listing consisting of user prims (<primitive: 11106>, etc.) with no comments, and is presented before the reader is even introduced to the necessary semantics. The book does end with a fairly good discussion of simulation and Smalltalk applications in simulation. Perhaps this discussion should have been presented much earlier.

All in all, I was disappointed. Given the great need for books and monographs on scientific and engineering applications of Smalltalk, perhaps I expected too much. In all fairness, the book is accompanied by an instructor's manual and code disks, which were not available in time for this review. Perhaps their presence would have presented the text in a different viewpoint. Future books on this topic should emphasize Smalltalk as a behavioral paradigm for computational simulation of physical processes. This important "forest" should not be hidden by "trees" of small details. ▣

Richard L. Peskin is Professor of Mechanical and Aerospace Engineering at Rutgers University and director of the CAIP Center Computational Engineering Systems Lab. He has been involved with engineering and scientific aspects of Smalltalk since 1984. In addition to doing research in computational fluid dynamics and non-linear dynamics, he is one of the designers of the SCENE (Scientific Computation Environment for Numerical Experimentation) system, a Smalltalk-based distributed computing environment that implements computational steering tools such as interactive scientific graphics and data management, automatic equation solvers, and mathematical expert systems.

Highlights

Excerpts from industry publications

CONCEPTS

. . . In most languages, learning to program means learning the syntax. Learning to program in Smalltalk, however, involves much more. The programmer must have a clear grasp of object-oriented concepts. In addition, Smalltalk's development environment strongly influences the entire approach to software creation. It is absolutely essential that the developer become familiar with the classes provided by the Smalltalk environment. Although this can take some effort, it's a prerequisite for developing more than the most trivial programs. Fortunately, this is an interesting activity and is one of the best ways to learn Smalltalk.

An earful of Smalltalk, John D. Williams, PCAI, 9-10/92

TASKS

. . . The tasks in an object-oriented effort are different. New tasks are required to identify, characterize and document objects. These tasks focus on identifying objects and the interactions required of these objects to provide a system that meets stated requirements. Object-oriented efforts, like other development approaches, need requirements and design specifications. Yet these documents localize around objects, and not functions or data. In addition, these specifications clearly delineate which components are reused from an in-house reusability library and which are developed from scratch to support the application at hand. Tasks associated with the construction of structure charts, data flow diagrams and other function- or data-oriented modules are obsolete and replaced with modeling approaches more in concert with object-oriented development.

Designing the object-oriented way, Ron Schultz, OPEN SYSTEMS TODAY, 7/20/92

END-USER DEVELOPERS

. . . No fundamental change in the pace of software development can occur until there is a significantly higher level of application development. In other words, end users must become developers. Object-oriented programming could allow end users to do just that. The ideal application development environment would consist of enormous libraries of prefabricated, modular program parts (super high-level objects). These modules could be configured and combined in virtually unlimited combinations to build complete applications across the entire spectrum of software use. Applications would be built exclusively in a high-level tool of this sort. Conventional code-level programming would focus on creating object components . . . . End users would have unprecedented programming opportunities.

The new shangri-la?, Joseph Firmage, SOFTWARE MAGAZINE, 7/92

BOOK REVIEW

Richard L. Peskin

OBJECT-ORIENTED ENGINEERING

by John R. Bourne

The subtitle of this book is *Building Engineering Systems Using Smalltalk-80.* It is to Bourne's credit that he addresses the important topic of engineering applications of object-oriented software systems. While simulation was a primary target of early object-oriented languages, such as Simula and original versions of Smalltalk, more recent activity in the subject area appears to emphasize business applications, data base applications, etc. If Smalltalk is to take its place alongside more commonly accepted languages, its success in scientific and engineering applications will have to be demonstrated on a much broader scale than is present today. Bourne's effort provides an important step in that direction, namely a book that addresses uses of Smalltalk in the engineering domains.

The author has made some valuable contributions to applications of Smalltalk in the college classroom, one example being his work on engineering tutorial systems implemented in Smalltalk. The book, however, is somewhat disappointing as a classroom tool or general resource for engineers who want to learn more about Smalltalk's potential for technical applications. The material is much too general in its treatment of actual engineering applications, yet at the same time contains too much code-level detail without providing sufficient preparation for beginners.

Part I is an overview of general concepts such as representation of physical processes in terms of objects and behaviors. A serious deficiency is the lack of historical perspective and presentation of important recent contributions in engineering applications of Smalltalk. Notably absent is any mention of the contributions of the (now defunct) Tektronix group. Applications such as INKA, a system that assists in instrument service, represent important real engineering Smalltalk projects. Also omitted are the contributions of Thomas et al. on the uses of Smalltalk in realtime instrumentation and control, work done at Rutgers on scientific data management, and other real-world cases discussed in recent journals and proceedings. Engineers need to be motivated by actual applications.

Turning to more specific issues, this reviewer would have liked to have seen more emphasis on behavioral paradigm, as opposed to software structural aspects (inheritance, etc.). Encapsulated behavior of objects is the crux of what Smalltalk has to offer engineering simulation. Bourne puts much emphasis on the MVC paradigm and attempts to draw real-world analogies. Not only is MVC out of date, but the author's analogies are somewhat questionable. My greatest criticism of this part, and of the book as a whole, is the emphasis it places on use of ACOM cards for the "pre-specification" of a Smalltalk design. Bourne goes so far as say that one must use 4x6 cards as opposed to 3x5 cards for writing down the desired classes, protocols, etc. This approach reflects the traditional "specification" approach to software, not the interactive prototyping style that is Smalltalk's forte. Although he references a 1986 paper by Cunningham and Beck as his rationale for emphasis on ACOM cards, my own reading of that paper was that cards were only a "literary aide" to help explain O-O concepts. The first part of the book ends with an overview of other O-O languages, in which the author does emphasize the importance of having a complete class library for a particular O-O environment to be of real benefit.

Part II concentrates on "tools," namely the Smalltalk language and environment. This section does not flow smoothly from topic to topic and I fear it will be difficult for beginners to follow. Smalltalk code examples are presented in numerous figures without proper preparation for the lay reader. Perhaps Bourne intended this section to be covered by additional classroom material. In addition to Smalltalk specifics, this section covers issues such as "look and feel" (but omitting that PPS release 4 does not have a complete native platform look and feel) and bit editors (without making clear that release 4 does not really support this and Pens as part of the system). As in Part I, great store is place on the ACOM card method and how to transfer information from the cards to the Browser. However, there are some useful pieces in this section. While the discussion on page 147 mixes animation with drawing, at least one is shown how to draw a line using PPS release 4. Chapter 8 concentrates on MVC. There is too much detail, particularly about the viewBuilder, and that level of detail is really not germane to the subject of engineering applications. It is interesting to note that the author's own code example for MVC illustrates the typical MVC problem; that is, where to put drawing methods. The "Counter" examples ParcPlace used to distribute would be better in this context. The author discusses the "Pluggable Gauges" package (from KSC), but doesn't refer to the active value concept that is central to that package and important to engineering applications.

Part III deals with engineering applications, which I found to be the most disappointing. Most of the discussion about exam-

TAKING EXCEPTION TO SMALLTALK *continued from page 1*

This article and its sequel next month present a Smalltalk implementation of exception handling. This month, we'll describe the system's interface and the machine-independent aspects of its implementation. Next month, we'll complete the picture by describing the V 286-specific implementation.

THE EXCEPTION HANDLING INTERFACE

At the heart of the exception handling system are the classes Signal and Exception. An instance of Signal represents an exceptional event that might occur and its most important methods, handle:do: and raise. Sending handle:do: to a Signal object registers a block that can be evaluated if that event occurs. For example, suppose OutOfBoundsError is a global variable that holds a Signal object. As the name implies, this signal is intended to signify out-of-bounds references in arrays and might be used in a method of class Array as follows:

```
checkFifthElement
    OutOfBoundsError
        handle: [ :exception | ^self handleException: exception]
        do: [ ^self at: 5]
```

The effect of handle:do: is to evaluate the second parameter (do: block), with the addition that a raised OutOfBoundsError will be handled by evaluating the first parameter (handle: block). So, as you might expect, evaluating #(1 2 3 4 5) checkFifthElement will return 5, but evaluating #(1 2 3 4) checkFifthElement will cause the block [:exception | self handleException: exception] to be evaluated. What happens next depends on Array>>handleException: It might define a default value for that array, prompt the user for information, or form some other appropriate response.

For this scheme to work, the system must use OutOfBoundsError to signify the out-of-bounds condition. This can be done by sending the raise message to OutOfBoundsError in the midst of at: (and methods like it), as follows:

```
at: anIndex
    <primitive: 60>
    (self outOfBounds: anIndex)
        ifTrue: [^OutOfBoundsError raise]
```

One interesting aspect of the handleException: message is its parameter exception, which is an instance of the class Exception. Each time a signal is raised, a new exception is created to objectify that fact. The exception is a convenient place to encapsulate information about both the signal and the context in which it was raised. Particular error information or a special error message can be associated with an exception by using variations of the raise message, in this case raiseWith: and raiseErrorString:, respectively. In this way, an exception handling block can learn a great deal about the error by accessing the exception, which allows it to respond more intelligently.

In addition, class Exception provides support for common exception-handling techniques, including the messages proceed, reject, restart, and return. When an exception proceeds, control resumes in the context where its signal was raised, and a value can be returned if desired. This is how a new default value can be defined for an array. Thus, handleException: could be implemented as:

```
handleException: anException
    anException proceedWith: 'Bob'
```

This will cause the string 'Bob' to be returned as the value for any index outside the array's bounds. In addition to proceed, you can send restart to an exception, which causes the handle:do: context to be restarted, or send return, which causes the handle:do: message itself to return, again with the option of returning a specified value. Finally, sending reject to an exception is a way of saying that the current handler can't solve the problem. The system looks for the next handle:do: context down the stack that can handle the signal and evaluates its handle: block. These possibilities are illustrated in Figure 1.

For the purposes of this example, we assume that Array>>foo is implemented as:

```
foo
    Transcript show: self checkFifthElement printString
```

Now, if #(1 2 3 4) foo is selected and evaluated, when fetchHandlerBlock returns, the context stack will be as shown in Figure 1, with the exception's instance variables signalContext and handlerContext referring to the indicated contexts.

There are several ways to define Array>>handleException:. One possibility is for it to proceed from the exception, as in

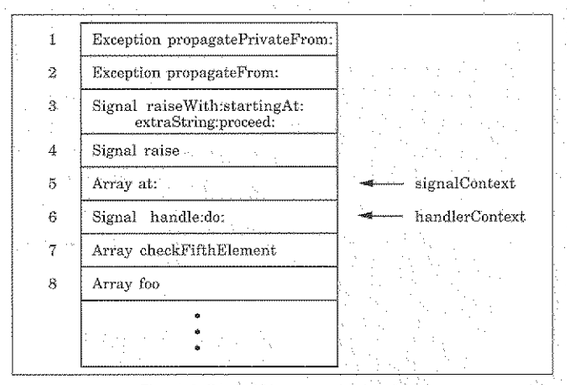


Figure 1. Stack during exception handling.

```
handleException: exception
    exception proceed
```

In this case, when the handle: block of handlerContext is evaluated, nil will be returned as the value of the Array>>at: message send, the fifth context on the stack, and execution will proceed in the sixth context. However, if handleException: is defined as

```
handleException: exception
    exception return
```

then nil will be returned as the value of the Signal>>handle:do: message send corresponding to the sixth context on the stack, and execution will proceed in the seventh context. Using restart, as in

```
handleException: exception
    exception restart
```

will cause the handlerContext, the sixth context on the stack, to be restarted from the beginning, in effect reevaluating the do: block. Finally, the exception handler may reject the Exception, as in

```
handleException: exception
    exception reject
```

In this case Exception>>propagatePrivateFrom: will be called again, but this time the search for a handler will proceed downward from the context just below the handlerContext, in this case the seventh one on the stack.

> Briefly speaking, exception handling is the provision for non-lexical flow of control in a program when something out of the ordinary (i.e., exceptional) occurs.

There is one final part of the system that interacts with exception handling, though it's not implemented in either of the above two classes. This feature is something called an unwind mechanism, which is a way for a programmer to ensure that certain actions are performed, even if a context is skipped during exception handling. For example, when an exception does a proceed, restart, or return, the flow of control jumps into lower contexts on the procedure's stack, and any higher contexts are removed from the stack without ever returning to them. This can be a problem: The contexts that were skipped might have performed some clean-up actions, such as closing files or releasing semaphores, if they'd been allowed to finish execution and return normally. Skipping these contexts during exception handling means skipping important clean-up jobs. The solution to this problem is to define a special method, whose purpose is to ensure clean-up blocks will be executed, even in the presence of exception handling. The name of this method in Smalltalk-80 is valueOnUnwindDo:. Assuming aCollection is defined, evaluating

```
[aCollection checkFifthElement]
    valueOnUnwindDo: [Transcript show: 'Time to clean up!']
```

will cause the first block, [aCollection checkFifthElement], to be evaluated. If aCollection has five or more elements, the value of the fifth element will be returned, and nothing more needs to be done. However, if aCollection has four or fewer elements, and if the exception handler for OutOfBoundsError causes control to return past the context of the valueOnUnwindDo: method (in effect skipping it), the second block will be evaluated, allowing any clean-up or finalization to be done. In Smalltalk-80, unwind blocks are even executed if they're skipped by a normal method return, because up-arrow is treated just like a return from an exception. In V 286, though, the meaning of up-arrow is hardwired into the virtual machine, so we can't duplicate this behavior.

THE MACHINE-INDEPENDENT IMPLEMENTATION

Although an implementation of exception handling inevitably delves into system-specific code, much of our solution is system independent. In fact, the same implementation of class Signal is used for Tektronix and Digitalk platforms (and potentially for ParcPlace), and most of class Exception is common as well. This section considers the system-independent aspects of the exception-handling package.

To begin with, there are a number of predefined signals, all of which are defined in the Signal class>>initialize method and accessible using messages to Signal. These basic signals include ones for unhandled exceptions and keyboard interrupts. In addition to these, a class variable called ErrorSignal is added to Object (just be careful how you add it!) and is accessible by using Object>>errorSignal.

To create a new signal, you send the message newSignal to an existing signal. So, for example, we could create the signal OutOfBoundsError by evaluating

```
OutOfBoundsError := ErrorSignal newSignal
```

either in a workspace or (more likely) in a class initialization method. The newSignal method creates the new object and sets its parent instance variable to the receiver. The parent variable in class Signal is used to provide more structure in signal handling. When a signal is raised, it can be handled by an exception handler for the signal, by one for the signal's parent or by one for any of the signal's ancestors. In this way, a programmer can define some general response for a tree of signals by registering a handler for the signal at the root. This response can then be specialized by registering more specific handlers for the signals further down in the tree.

Once a signal has been defined, sending it handle:do: registers an exception handler for it. The code for handle:do: is

the browser. This is precisely why more recent Smalltalk programming environment extensions come equipped with mechanisms and tools that explicitly enable designers to package the presentation of a class and its interfaces to casual users.

I do not want to digress into a discussion on the merits of recent additions to Smalltalk programming environments. (I am absolutely convinced of their utility.) Nor do I particularly want to defend Smalltalk against languages with explicit support for public and private declarations (which have problems in actual use). However, developers of these newer Smalltalk environments have recognized the danger of information overload. Without removing detail, it may be difficult to discover the essence of a class.

We often create an instance and only use a fraction of its class's features. And we are completely content to do so. I strongly advocate a written textual description of a class, describing the typical and most important patterns of use. Describe the essential 20%, 50%, or 80% (your percentage will vary depending on how full-featured a class is and how much exploration a programmer makes) in a few short paragraphs. Accompany this description with a few pictures describing typical object-interaction sequences. Leave the rest for me to discover by either reading through a more detailed class-design document or by exploring your code and comments. If you are trying to leave a helpful trail for users, embed a typical object-creation message with appropriate arguments inside a comment within an instance creation method. More elaborate examples can be developed with detailed comments, either to be filed into an image or executed.

SPEND TIME ON WHAT MATTERS

Not every class is worthy of the same amount of attention. A class of limited utility, intended to be seen by a very small audience, only deserves light treatment. I am not a proponent of mandating equal discussion for all classes. That leads to either lots of useless boiler-plate documentation or developer mutiny. Instead, spend the time creating a well-considered discussion for classes that provide broadly useful functionality or are central to your design.

Complex classes that require a lot of set up or have highly stylized patterns of usage demand extra attention. From an external viewpoint, I need to know common patterns of usage, as well as how to diagnose an object that's broken and not functioning as expected. We creators of initial designs often don't realize how easy it is for someone else to misinterpret our work. So this kind of discussion is definitely worthwhile, if only to get an idea of potential hot spots.

MAKING THE CONNECTIONS

It is relatively easy to produce documentation for a class intended to be used in isolation. It is much harder to describe classes that are part of a larger framework and intended to be used in conjunction with a number of collaborators. To use a framework requires understanding how objects interact, what role each object plays, and when and how objects should be created and used.

A description for a framework of interacting classes must not only cover the central classes, but also establish a clear model of how these classes are intended to work together. This year's OOPSLA conference had a refreshing paper by Professor Ralph Johnson that explained his process describing a graphical editor framework in Smalltalk, called HotDraw. HotDraw was originally developed by Ward Cunningham and Kent Beck. In five pages of text, Ralph described the central ideas behind HotDraw and documented some common patterns of key objects and their interactions. A nice touch was clear references to the next layer of detail as well as pointers to related concepts for each pattern of use.

Simple, helpful descriptions of object-interaction patterns are straightforward reading. They require that the author has a clear vision of the core ideas of a framework and a simple, if not terse, writing style.

It reminded me of the *Choose Your Own Adventure* books my kids used to read. After one or two pages, you were asked a question. Depending on your answer, you were directed to one of two pages. You could read the entire book and get several different stories, each with different endings. My kids were never satisfied until they had explored all possible paths.

Documentation of interlocking classes of objects needs this touch. First you need a description of core concepts. Then you need to tour key interactions at your own pace, allowing you to discover and explore according to your personal choices. Descriptions should let you navigate, point you to more detail (if you want it), and let you move on (should you want to broaden your understanding).

CONCLUSION

New, useful ways for describing classes of objects and groups of cooperating objects are active research topics. There's plenty of room for formal techniques as well as informal descriptions. What I constantly strive for are pragmatic ways to impart design insight to users.

I don't want you to leave with an impending sense of doom or writer's block. I don't like writing reams of paper that no one reads. And I won't recommend that you take extraordinary measures nor do what I personally am not willing to do myself.

I especially want to appeal to you cynics who might be thinking as you read this, "But she's a writer. Of course she can recommend we do these things. Writing comes naturally to her." Writing is definitely not a natural act for me. I have to struggle to write concise, precise documentation. But as a user of some pretty nicely described systems, I encourage you to perform an enormous service to your users. Take some time to describe how to properly use your classes. ▩

Rebecca Wirfs-Brock is Director of Object Technology Services at Digitalk and co-author of DESIGNING OBJECT-ORIENTED SOFTWARE. *Comments, further insights, or wild speculations are greatly appreciated by the author. Rebecca can be reached via e-mail at rebecca@digitalk.com. Her U.S. mail address is Digitalk, 7585 SW Mohawk Street, Tualatin, Oregon 97062.*

PUTTING IT IN PERSPECTIVE

Rebecca Wirfs-Brock

Describing your design

Objects can be simplistic and passive, holding on to small pieces of information, or they can be busy and active, serving an important role in framing the overall architectural structure of an application. The possibilities for what an object can represent are limited only by human imagination. In this column I want to explore some effective techniques for describing classes so they can be understood, used, and refined by others. You, the author of a class or a group of collaborating classes, know how you intend them to be used. How can you effectively impart this knowledge to others? However you describe a class, your original design intent will be mulled over by different people, each with a slightly different set of expectations, needs, and experiences.

There are basic things that need to be said about any class. These essentials cover roughly 50% of the issues, which I'll cover first. Then I want to explore the remaining 50% that are often left unsaid.

COVERING THE BASICS

Each class you construct in your design has a specific purpose. You know what the class was intended to do and probably what it was never intended to do. (It is easy for someone to torture your code in ways you never dreamed of, but I don't know how to solve that problem.) You also know whether your creation is of major or minor importance, whether you have a polished implementation, or whether you have left room for improvement. The exact details you need to communicate vary depending upon the role of the reader. Different information and levels of detail are needed by:

- a programmer wanting to use this class in a program
- a developer creating a subclass to add new functionality or override existing behavior
- someone adding new functionality to your class
- anyone trying to understand a class inheritance hierarchy
- a tester developing test suites
- someone fixing a programming error

When we describe our classes and our applications, we need first to provide a global context (a road map of the territory). This provides a broad view, allowing readers to understand how individual classes fit into the overall fabric of your design. This should then be augmented by a consistent discussion of classes from both an exterior (usage) and interior (implementation) perspective. Arguably, all potential readers of class documentation need a basic understanding of how a class should be used.

Let's concentrate on what informed class users need to know. At first glance, to use a class, a programmer needs to know:

- what the class was designed to do and not to do
- ways to create an instance of that class and, subsequently, how it typically is used
- what it depends on, including other objects, global states, or host–operating system features
- where to look for further details

Subclass developers need this information to ensure that their new addition follows the expected patterns of behavior defined by its superclass. They should not fix one problem only to break pre-existing contracts with all current users of the class. They need even more details than users, but all proceed from these basics.

Not all basic information is gleaned by poring over a class-browser reading code. Some have claimed that Smalltalk's programming environment eliminates much need to describe this kind of information, but this is just another rather lame argument that XXX code (replace XXX with your favorite programming language) is self-documenting.

Learning an application by reading code and performing experiments can take a long time and often isn't the most effective way to transfer knowledge. We designers and implementors of classes should explain how to create and use our objects. Documentation should supplement a programmer's ability to find and use the right classes for the job.

From an exterior view, I certainly need to know less than someone who is intending to modify, extend, or create a subclass. I want you, the designer, to hide those things I shouldn't care about. I really don't want to concern myself with any of the object's instance variables, unless you explicitly choose to give me access to them. I also don't care about implementation details encapsulated within methods. And I certainly don't care about code that is private, intended to be executed by sending messages to self. So please label those private, internal details as such. Your chosen method partly depends on your Smalltalk environment, and partly on style guidelines used within your organization.

Understanding how to create and use an object can sometimes become confused by all that wonderful detail exposed by


```
handle: handlerBlock do: doBlock
    "Evaluate doBlock. If all goes well, return its value. If an exception
    occurs then the returned value could be generated by evaluating
    returnBlock."
    | returnBlock |
    returnBlock := [ :value | ^value].
    ^doBlock value
```

This method's most significant role is as a placeholder. Its basic function is simply to evaluate its second parameter, the do: block. But it also marks a place on the context stack so the system can find an appropriate handler when an exception occurs. How this happens will be explained next month when we consider Exception>>fetchHandlerBlock:. The block stored in the returnBlock temporary variable is used to make implementing Exception>>return easier.

The only other method we mentioned for class Signal was raise. As we said before, there are actually many variations of the raise message, depending on whether the exception handler can proceed through the exception, whether there's a parameter or error string needed, and so on. All these raise combinations call the same private method, which is Signal>>raiseWith:startingAt:extraString:proceed:. This is implemented as follows:

```
raiseWith: parameter startingAt: context
extraString: str proceed: aBoolean
    "Create a new exception and have it look for handlers
    starting at context."
    | exception |
    exception := self newException
        signal: self
        parameter: parameter
        extraString: str
        proceedBlock:
            (aBoolean
                ifTrue: [[:value | ^value]]
                ifFalse: [nil]).
    ^exception propagateFrom: context
```

This method creates a new instance of Exception, passing the signal as one of the parameters in the creation message. In addition, if aBoolean is true, the signal is "proceedable", which means that the handler is allowed to send the exception the proceed message, in effect declaring the error completely resolved and causing a return from the raise message send. If it is proceedable, the new exception will be passed the block [:value | ^value]. Like returnBlock in the handle:do: method, the block here simplifies our implementation, in this case making Exception>>proceedDoing: much simpler. Finally, this new exception is sent the message propagateFrom: with the context passed in as a parameter. This begins the process of finding a handler for the exception.

Exceptions have five instance variables: signal, parameter, extraString, proceedBlock, and handlerContext. The first four are set by the signal:parameter:extraString:proceedBlock: message, which is sent by a signal when the exception is created. The value of proceedBlock, if it isn't nil, is the [:value | ^value] block we saw above. After creating a new exception, a signal sends the propagateFrom: message, which in turn calls the propagatePrivateFrom: method. In addition to error handling, propagatePrivateFrom: sends the message fetchHandlerBlock: to find the right handler for the exception (in the process, it sets the instance variable handlerContext to the appropriate handle:do: message's context) and evaluates that handler. The implementation of fetchHandlerBlock: is described in next

continued on page 14...

GUIs

Greg Hendley and Eric Smith

Significant supported events in Smalltalk/V PM as illuminated by Window Builder

If you have used Window Builder by Cooper & Peters, then you have taken advantage of its fill-in-the-blank way of writing when:perform: statements for the open method. You have probably noticed there are more events than you thought you needed. You may have even asked yourself, "Should I be using these events and, if so, how?"

In this installment of GUI Smalltalk, we will discuss some of the significant supported events for the subpanes and controls directly supported by Window Builder. This is not intended to be an exhaustive discussion of every event; it will, however, get the adventurous off to a good start.

We decided classes that implement supportedEvents would be the most interesting to look at. The remaining classes should inherit their superclasses' behavior. We will discuss each class in turn, including some significant supported classes.

TopPane

Nearly all windows involve some kind of TopPane, which is usually the window containing all the other controls. TopPanes support a number of events that no other kind of window is interested in.

- **#validated.** This event is generated as the final act in opening a TopPane. When this occurs, the pane represents a valid Presentation Manager (PM) window. This event seldom requires a handler. However, in some rare instances, it provides an opportunity to do any necessary twiddling of the PM frame window after it has been opened but before any of the children have been opened.
- **#activate.** When a frame window becomes the 'active' window (i.e., it is selected, given the active window border color, and the input focus), the window message WM_ACTIVATE is sent along to the PM frame window. In Smalltalk, this results in the #activate event. A newly opened window usually becomes the active window, so this happens when the window is opened as well as each time the frame window is activated.
- **#menuBuilt.** The #menuBuilt message is generated after the menu bar has been created but before children are opened or the TopPane validated. If you are using WindowBuilder, this event is unlikely to occur. Cooper & Peters have circumvented the normal menu bar creation methods in their open methods. Ordinarily, this event might be used to initialize the enable/disable state of the various menu choices, add custom menus, etc. When using WindowBuilder, these sorts of activities can be performed in the #initWindow method.
- **#close.** Whenever TopPane, not ViewManager, receives the message #close, it will generate the event #close before taking any action. If there is no handling method, or the handling method returns nil, then the close operation will proceed normally. Otherwise, no panes will be closed. Handlers for this event are quite common, particularly if dependents are used. This provides the ideal place to clean up dependents, PM resources, and other potential garbage as the window disappears.
- **#help.** The #help event occurs when help is called for via the F1 key. Using the help menu (should one be available) will not generate this event. The handling method may do whatever it pleases by way of providing help (e.g., toss up a dialog, open another application, put up a message box). If there is no handling method, or the handler returns nil, then the problem will be passed along to the PM help manager. Note that if you have a HelpManager defined for a window as well as a handling method for the help event, then unless the handling method returns nil, the PM help manager will not come up when F1 is pressed.
- **#timer.** This event will be generated whenever the frame window receives the window message WM_TIMER. This only occurs in special circumstances beyond the scope of this column.
- **#opened.** This event is a red herring. It won't hurt a TopPane to have a handler for this event, but that handler will never be activated because TopPanes don't generate this event.

DialogTopPane

DialogTopPanes behave just like TopPanes in most respects (including those having to do with generating events). All the events described for TopPane above are inherited, except that those having to do with the menu bar will not be given a chance to occur. One additional event is generated by dialogs:

- **#opened.** After a dialog is built, but before processing begins, the event #opened occurs. This provides the owning ViewManager with the opportunity to fill in entry fields, initialize button choices, etc.

you are manipulating the objects but not asking them to do anything. For instance, if you designed a remote object system where transparent copies of objects were transmitted over a network, you might store the objects in an IdentitySet. If you transmitted two objects that were = but not ==, and later changed one of them, storing them in an IdentitySet would ensure that they were different objects on the remote systems.

Bag

Instead of discarding duplicate elements like Sets, Bags count them. Executing this code:

```
| s |
s := Set new.
s addAll: #(a a b b c).s
size
```

returns 3. Changing it to a Bag:

```
| b |
b := Bag new.
b addAll: #(a a b b c).
b size
```

returns 5.

Use Bags anywhere you want a quick implementation of includes—that is, when you don't care about the order of elements and you need a compact representation of duplicate elements.

Bags are not used anywhere in the ParcPlace release 4.1 image or in Smalltalk/V Mac 1.2. The only time I can remember using Bags is in Profile/V. Every time I take a sample, I put the program counter in a Bag. When I display the profile, I map the stored program counters back to source statements, giving the user profiling at the level of individual statements.

CONCLUSION

The Collection classes are one of the most powerful parts of the Smalltalk system. Choosing the right collection for a circumstance has a dramatic influence on the behavior and performance of your system. I have tried to lay out what each major collection class does, what it is good for, what to watch out for, and how it is implemented.

I am amazed at the richness of this seemingly simple set of classes. Originally, I thought I would have to stretch to get enough material for just one column. After two columns that have covered the major issues in using collections, there is still more to be written. I'll give it a rest for now, however, and go on to something else—I'm not sure what just yet. If you have any ideas call me at 408.338.4649 or fax me at 408.338.1115. ▩

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

cording to the order in which they were added, SortedCollections rely on a two-argument block to determine, pairwise, the order for elements. This block defaults to [:a :b | a <= b], so simple SortedCollections sort their elements from lowest to highest.

One thing to watch out for when using SortedCollections is sending them add: when you don't have to. add: does a binary search of the collection, moves all of the elements after the added object down one, and inserts the added object. Moving the elements to make room takes time proportional to the size of the collection. If you know you are going to be adding several elements at once, use addAll:, which will stick the new elements at the end and resort the entire collection. Here is a method for comparing time spent using these two methods (notice that I don't hold myself to the same coding standards in workspaces):

```
| sc r t1 t2 |
sc := SortedCollection new.
r := Random new.
t1 := Time millisecondsToRun:
    [1000 timesRepeat: [sc add: r next]].
sc := SortedCollection new.
t2 := Time millisecondsToRun:
    [sc addAll: ((1 to: 1000) collect: [:each | r next])].
'Add: ', t1 printString, ' addAll: ', t2 printString
```

Executing this results in 'Add: 10725 addAll: 1386'.

String

Strings in Smalltalk are like Arrays whose elements are restricted to Characters. Strings are byte-indexable for compactness. They redefine the indexing methods to convert from 8-bit numbers to characters and vice versa:

```
String>>at: anInteger
    ^Character value: (super at: anInteger)

String>>at: anInteger put: aCharacter
    ^super at: anInteger put: aCharacter asciiValue
```

It is common to use , to concatenate Strings. You can use , to concatenate any two sequenceable collections (OrderedCollection, Array, RunArray, and so on). Less common is the use of the other collection methods with Strings. You can capitalize all the characters in a String with collect:

```
asUppercase
^self collect: [:each | each asUppercase]
```

Interestingly, even the ParcPlace release 4.1 image implements this method with five lines containing an explicit loop and indexing.

Digitalk's String class is implemented with the simple model described here. ParcPlace has a much more elaborate implementation that takes care of multibyte characters and different character sets on different platforms, even for odd characters. The design requires six classes for strings and three more for symbols.

Symbol

Symbols behave in most ways like Strings, except that if you have two symbols containing the same characters, they are guaranteed to be the same object. So while String>>= takes time proportional to the length of the strings, Symbol>>= takes constant time:

```
Symbol>>= anObject
    ^self == anObject
```

To preserve uniqueness, Symbols cannot be changed once they are created. at:put: is overridden to raise an error.

Like Interval, because Symbols don't respond to at:put:, they override species. Symbol>>species returns the class String. Thus, executing "#abc , #def" returns 'abcdef', a String, not a Symbol.

If you are programming in Smalltalk/V, be careful of creating too many symbols. There is a limit of 2^16 Symbols. While this may seem like a lot, after you have created many new methods and used Symbols for indices in several places, it is very possible to run out of Symbols. The scrambling you have to do to climb out of the "limited Symbol pit" is not pretty.

A last oddity of Symbols and Strings is the asymmetry of =. "'abc' = #abc" returns true because the String receives the message and successfully checks to see that the characters in the receiver are the same as those in the argument. "#abc = 'abc' " returns false because the two objects are not identical. I can remember long debates at Tektronix over the propriety of this strange fact. The upshot of the debates was that it's regrettable things work this way, but the alternatives are all less attractive for one reason or another.

Sets

Sets are dynamically sized collections. They respond to add: and remove: but, unlike OrderedCollections, they don't guarantee any particular ordering on the elements when they are used later (e.g., by do:). Sets also don't have any indexed access (no at: or at:put:).

Sets implement includes:, add: and remove: efficiently by hashing. The element to be added is sent hash, and that value is used modulo the size of the storage allocated for the Set as the index to start looking for a place to put the element (or remove it). Note that storage for a Set will contain more indexed variables than the Set has elements, so hashing is likely to encounter an empty slot. The Set contains an instance variable, tally, which records how many of the slots are filled. Set>>size just returns tally.

You can eliminate duplicates from any collection (albeit while losing its ordering) by sending it asSet.

IdentitySet

Sets use = to determine if they have found an object. IdentitySets use ==. They are useful where the identity of objects is important. Most applications are in meta-object code, where

SubPane

SubPane is included even though it is an abstract class. Many normal behaviors are described in this class. We will take advantage of inheritance in our descriptions and only deviations and additions will be described for subclasses.

- **#display.** While SubPane supports this event, it is only received by GraphPane. So for all other subclasses, unless you write a method that sends event: #display, you can disregard this event.
- **#resize.** This is sent after PM has resized a top pane (or other subclasses of ApplicationWindow). Most applications have no need for this event. Possible exceptions are special uses of GraphPane and GroupPane. Most resizing is handled with the normal get contents and display methods. This is supposedly one of the advantages of using an existing windowing system such as PM.
- **#getPopupMenu.** This event normally occurs as a result of the mouse button2 click. No surprise here.
- **#getMenu.** This event is usually not sent if the window was built using Window Builder. The exception (there's always an exception) is when the pane looks for its pop-up menu. If it can't find one, it looks for its regular menu to use for a popup. Therefore, it is your choice to use this or the previous event for your pop-up menus. Proper discussion of menus would require its own column.
- **#getContents.** Now we are back on familiar ground. This event is sent whenever a subpane is opened. It is used to set the text of a text pane, list of a list pane or combo box, and label or text for other controls. This setting is normally done using the method contents:. It is also sent as part of the restore and update methods for many classes.
- **#help.** This is normally sent when the F1 key is pressed. Not all subclasses receive this event.

TextEdit

- **#textChanged.** This event is sent each time a character key, backspace, or delete is pressed. Think about whether you want to respond. This event will be sent frequently if entire paragraphs are being typed.
- **#horizScroll.** You normally will not care about this event, which is sent when you scroll using the horizontal scroll bar. It also happens with automatic scrolling, which occurs when you type past the pane and word wrap is off.
- **#vertScroll.** This is similar to horizScroll.
- **#help**, **#getPopupMenu**, and **#getMenu**. None of these are received.

TextPane

TextPane inherits events from TextEdit. It also adds one event:

- **#save.** This is sent through selecting the "save" item in the pop-up menu for TextPane.

ListBox

- **#charInput.** Most Smalltalkers do not use this event; they use the event #select, which happens when a character is typed. If a character is the first character of one of the items, that item is selected.
- **#drawItem** and **#highlightItem.** Seldom used by most Smalltalkers, these are sent only when a user-drawn item is included in the list of items. This deserves its own column and will not be discussed here.
- **#select.** This event occurs when an unselected item is selected, not when a selected item is re-selected. It also occurs when an item is selected by typing its first character.
- **#doubleClickSelect.** This event happens whenever an item is double clicked. Behavior is the same whether or not the item was already selected.

ListPane

Although neither super- nor subclass of ListBox, ListPane behaves similarly. The exception is as follows:

- **#select.** This event occurs when selecting an item that is already selected.

ENTRYFIELD

Entryfield is the Smalltalk class representing one-line entry areas commonly seen littered about dialogs, although they may be used in any window. Most of Entryfield's interesting behavior can be used by paying attention to only two events:

- **#getContents.** As with most other panes, this event is generated by an Entryfield when it first comes up. It provides a nice opportunity to initialize the text contained in the entry field before the user gets to it. This is done in the handling method by sending #contents: to the pane with an appropriate String as an argument.
- **#textChanged.** Any time the contents of an Entryfield are changed, the #textChanged event is generated. It doesn't matter how the change originated; whether the user typed in more characters or somebody sent #contents: to the Entryfield, a #textChanged event is generated. This means that setting the contents of an Entryfield in the handler for a #textChanged generated by that Entryfield will lead to infinite recursion.

ComboBox

- **#textChanged.** Be careful about using this event as a trigger for other activities. We recommend you save the new text somewhere or note that the text is changed. One thing you do *not* want to do is update. This will create a circularity. The event #textChanged is sent in response to several activities: once when contents is set and twice when you type the first letter of one of its list elements. It is *not* sent when you type any other character. It *is* sent when

you press the pull-down button and when you select an item from the list.

- **#charInput.** This happens whenever *any* character is typed. Notice the difference between this and the previous event. A character can be typed without being entered into the text part of the combo box.
- **#select.** This event occurs at peculiar times the way #textChanged does. It is sent twice when text is in the entry field part and the list is pulled down. It is sent once when no text is in the entry field part and the list is pulled down. It is *not* sent when an item is selected that matches the text in the entry field part. It *is* sent once when an item is selected that does not match the text in the entry field part.
- **#doubleClickSelect.** This event does not happen for the ComboBox.
- **#drawItem.** This event occurs when a user-drawn item needs to be drawn. Most Smalltalkers will not use this event.
- **#highlightItem.** This event occurs when a user-drawn item needs to be highlighted. Most Smalltalkers will not use this event.
- **#listVisible.** This happens when you press the pull-down button. Most Smalltalkers will not use this event.

BUTTON

Button is the superclass of several kinds of controls that get clicked. Nearly all of them generate events, which are expected to be handled in similar ways.

- **#getContents.** This occurs when the pane first comes up. It can be used as an opportunity to set the contents of the button. For most kinds of Button, the #contents: message expects a String as an argument. This String will become the label for the button.
- **#clicked.** Any time a Button is pressed, the #clicked event occurs. For instances of Button, all you need to know is that the Button was pressed. For toggle-type buttons, the action of your handler may depend on whether the button was clicked on or off. This can be determined by sending the message #selection to the button. The Boolean returned will reflect the state of the button.

DrawnButton

The class DrawnButton represents a fairly special subclass of Button. It isn't like the others in that it has no predefined look. Instead, the owning window (or, in our case, the ViewManager) is expected to draw whatever it wants on the button's graphics context.

- **#getContents.** This event occurs when the pane first comes up. It may be used as an opportunity to provide the pane with a Bitmap, which it will draw on itself. DrawnButtons expect a Bitmap as an argument for the #contents: message.
- **#drawItem.** Any time a DrawnButton pane that does not have a Bitmap is asked to display, it will generate this event. When the handling method gets control, the DrawnButton pane will have a valid graphics tool. The handler method may then ask for its pen and draw whatever it wants on it. Note that this event also occurs as a result of the button being clicked.
- **#hightlightItem.** This message is generated as a result of pressing a DrawnButton. The underlying PM window messages inform as to whether highlighting is to be added or removed. Alas, by the time we reach the event level, this information has been lost. As with #drawItem, the graphics tool of the DrawnButton in question is valid while this event is processed.

SpinButton

Admittedly, this class is not directly supported by Window Builder. It is included in the standard image and can be added to Window Builder as a custom pane.

- **#getMenu, #getPopupMenu,** and **#help.** None of these are received.
- **#textChanged.** This is an unusual event in the number of times it occurs for a given action. It is sent once for each character typed. It is normally sent once when the up or down button is pressed. When there is text in the entry field that does not match any of its enumerated values, and the up or down button is pressed, the event happens twice. It happens once when the backspace key is pressed and twice when the delete key is pressed.
- **#up.** This event is sent when the up button is pressed. Normally, you would only look at the #textChanged event.
- **#down.** This event is sent when the down button is pressed. Normally, you would only look at the #textChanged event.
- **#getContents.** This event is ignored if the spin button is numeric. When the spin button is non-numeric, it expects to be told its list of enumerated values.

ScrollBar

Scrolling, with or without the scroll bar control, deserves more space than we can give here. We can, however, point out a few features.

The following events occur as a result of pressing the arrows, clicking in the blank areas, or moving the tab: #nextPage, #prevPage, #nextLine, #prevLine, #sliderPosition, #sliderTrack, and #endScroll.

The following events do not occur: #getMenu, #getPopupMenu, and #help.

#getContents occurs in the same manner as for most sub panes, but scroll bars do not know the method contents:. Instead, they use position:. ▣

Greg Hendley is a member of the technical staff at Knowledge Systems Corporation. His OOP experience is in Smalltalk/V(DOS), Smalltalk-80 2.5, Objectworks Smalltalk Release 4, and Smalltalk/VPM. Eric Smith is also a member of the technical staff at Knowledge Systems Corporation. His specialty is custom graphical user interfaces using Smalltalk (various dialects) and C. The authors may be contacted at Knowledge Systems Corporation, 114 MacKenan Drive, Suite 100, Cary, NC 27511, or by phone, 919.481.4000.

initial allocation, the space overhead and its effect on the storage manager can be significant. I have heard stories of programs speeding up by a factor of 60 just by replacing OrderedCollection new with OrderedCollection new: 1 at the right spot. Gather statistics on the number and loading of your OrderedCollections to determine if this optimization will help you.

Another performance implication of using OrderedCollections is the level of indirection required to access elements. at: as defined in Object just invokes a primitive to index into the receiver's indexed instance variables. To implement at: and at:put:, OrderedCollections have to take first into account:

```
OrderedCollection>>at: anInteger
    anInteger > self size ifTrue: [self error: 'Out of bounds']. ^
    super at: anInteger + first - 1
```

RunArray

RunArrays have the same external protocol as OrderedCollection, but they are optimized for storing collections in which the same object is added consecutively many times. Rather than just store the objects one after the other, RunArrays store two collections: one of the objects in the collection, the other the number of times the object appears (Figure 4).

Each entry in a RunArray requires two object references. RunArrays require storage related not to the number of elements in the collection, but to the number of times adjacent objects are different. In the worst case, RunArrays require twice as much storage as an OrderedCollection.

Indexing into a RunArray is potentially an expensive operation, requiring time proportional to the number of runs. Here is an implementation of at:

```
RunArray>>at: anInteger
    | index |
    index := 0.
    1 to: runs size do:
        [:each |
        index + (runs at: each) >= anInteger
            ifTrue: [^values at: each].
        index := index + (runs at: each)]
```

This simple implementation makes code like:

```
1 to: runArray size do: [:each | runArray at: each]
```

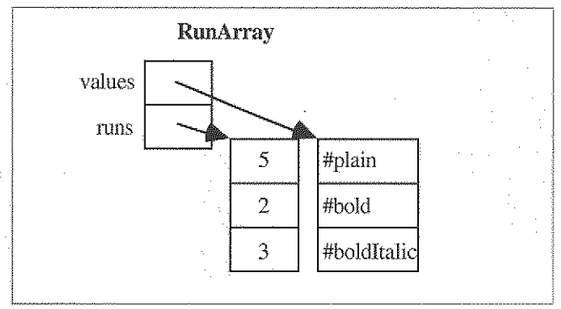


Figure 4. The result of RunArray new addAll: (plain plain plain plain plain bold bold boldItalic boldItalic boldItalic).

take time proportional to the number of runs multiplied by the number of elements in the collection. Because the access pattern for RunArrays usually marches along the collection from first element to last, RunArrays cache the beginning of the run in which the last index was found. Looking up the following index only requires checking to make sure that the new index is in the same run as the old one:

```
RunArray>>at: anInteger
    ^anInteger >= cachedIndex
        ifTrue: [self cachedAt: anInteger]
        ifFalse: [self lookUpAt: anInteger]

cachedAt: anInteger
    anInteger - cachedIndex > (runs at: cachedRun)
        ifTrue:
            [cachedIndex := cachedIndex + (runs at: cachedRun).
            cachedRun := cachedRun + 1].
    ^values at: cachedRun

lookUpAt: anInteger
    | index |
    index := 0.
    1 to: runs size do:
        [:each |
        index + (runs at: each) >= anInteger
            ifTrue: [^values at: each].
        index := index + (runs at: each)
```

With this implementation, an access pattern like the one above will now be slightly slower than the equivalent OrderedCollection because of the overhead of checking for the common case. Accessing the RunArray in reverse is now proportional to the number of runs squared.

Interval

Another kind of run-length encoded collection is Interval. An Interval is created with a beginning number, an ending number, and an optional step number (one is the default). #(1 2 3 4) and Interval from: 1 to: 4 are equivalent objects for most purposes. Number>>to: and to:by: are shorthand for Interval class>>from:to: and from:to:by:.

Intervals are commonly used to represent ranges of numbers, such as a selection in a piece of text. A common idiom is using an Interval with collect:.

```
foo
    ^(1 to: self size) collect: [:each | each -> (self at: each)]
```

Species is sent to an object when a copy is being made for use in one of the enumeration methods collect: and select:. The default implementation in Object just returns the class of the receiver. SequenceableCollection implements collect: and select:, and expects the result of self species to respond to at:put:. Since Intervals don't respond to at:put:, they have to override species to return the class Array.

SortedCollection

Another dynamically sized collection is the SortedCollection. Unlike OrderedCollections, which order their elements ac-

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS MAIL PERMIT NO. 279 DENVILLE NJ

POSTAGE WILL BE PAID BY ADDRESSEE

The Smalltalk Report
Subscriber Services Dept SML
PO Box 3000
Denville NJ 07834-9821

Smalltalk Object Database Support

Integrated garbage collection of persistent Smalltalk objects • Server-based gateway toolkit and relational gateways • Server-based active object manipulation language • Cooperative client/server support

Please send me information on **Smalltalk Object Database Support** and

☐ Keep me informed of future product announcements
☐ Have a Servio representative call:

Name: _____ **Title:** _____
Company: _____
Address: _____
City: _____
Phone: _____ **State:** _____ **Zip:** _____

SERVIO

GemStone, the ODBMS for C, C++ and Smalltalk from the Object Technology Company

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS MAIL PERMIT NO. 475 ALAMEDA, CA

POSTAGE WILL BE PAID BY THE ADDRESSEE

SERVIO CORPORATION
950 MARINA VILLAGE PARKWAY
SUITE 110
ALAMEDA, CA 94501

The Smalltalk Report

☞ **Provides objective & authoritative coverage on language advances, usage tips, project management advice, A&D techniques, and insightful applications.**

"If you're programming in Smalltalk, you should be reading The Smalltalk Report"

❑ **Yes, I would like to subscribe to The Smalltalk Report** Date_____

❑ 1 year (9 issues)
- ❑ Domestic $69.00
- ❑ Foreign $94.00

❑ 2 year (18 issues)
- ❑ Domestic $128.00
- ❑ Foreign $178.00

Method of Payment
- ❑ Check enclosed (payable to **The Smalltalk Report**)
- ❑ Bill me
- ❑ Charge my: ❑ Visa ❑ Mastercard ❑ Amex

Card No. _____
Exp. Date _____
Signature _____

Name _____
Title _____
Company _____
Address _____
City _____
State _____
Zip _____
Country _____
Phone _____

1. **Which dialect of Smalltalk do you use:**
- ❑ Smalltalk V
- ❑ Smalltalk-80
- ❑ Other _____

2. **What is your involvement in software purchases for your department/firm:**
- ❑ Recommend Need
- ❑ Specify Product
- ❑ Make Purchase
- ❑ None

3. **Which operating system supports your software:**
- ❑ UNIX
- ❑ DOS
- ❑ OS/2
- ❑ Windows
- ❑ Other _____

4. **What is your company's primary business activity:**
- ❑ Computer/Software Development.
- ❑ Manufacturing
- ❑ Financial Services
- ❑ Government/Military/Utility
- ❑ Educational/Consulting
- ❑ Other _____

5. **For how long have you been using Smalltalk:**
- ❑ Less than one year
- ❑ 1–3 years
- ❑ 3+ years

A member of the Object Marketing Network

fax to 212/274-0646

E2LG

SIGS PUBLICATIONS

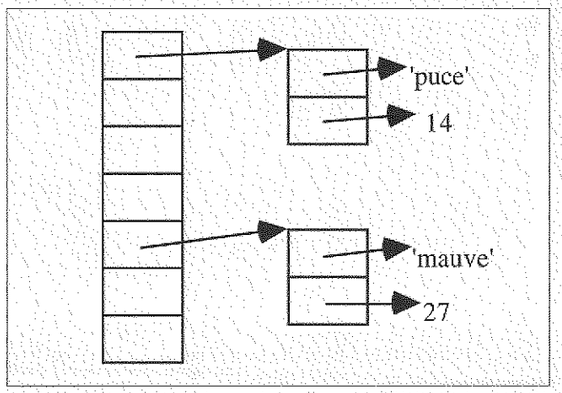


Figure 1. A typical Dictionary.

think of them as associations I use the message "associations" to get a set of associations I can operate on unambiguously.

When a Dictionary looks up a key it uses = to determine if it has found a match. Thus, two strings that are not the same object but contain the same characters are considered to be the same key. This is why when you reimplement =, you must also reimplement hash. If two objects are =, they must have the same hash value.

If you read your Knuth, you will see that hashed lookup takes constant time—it is not sensitive to the number of elements in the collection. This mathematical result is subject to two pragmatic concerns, however: hash quality and loading. When you send hash to the keys you should get a random distribution. If many objects return a number that is the same modulo the basic size of the Dictionary, then linear probing degenerates to linear lookup. If most of the slots in the Dictionary are full, the hash is almost sure to return an index that is already taken and, again, you are into linear lookup. By randomizing the distribution of hash values and making sure the Dictionary never gets more than 60% full, you will avoid most of the potential performance problems.

IdentityDictionary

IdentityDictionaries behave like Dictionaries except that they compare keys using == (are the two objects really the same object?). IdentityDictionaries are useful where you know that the keys are objects for which = is the same as == (e.g., Symbols, Characters, or SmallIntegers).

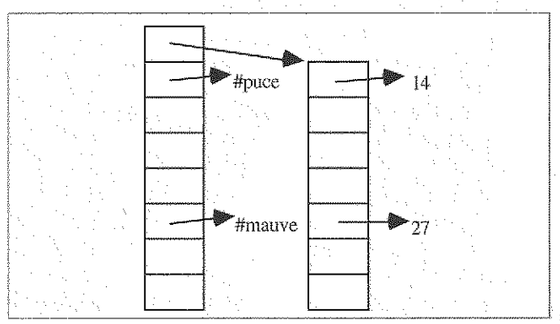


Figure 2. A typical Identity Dictionary.

Instead of being implemented as a hash table of associations, IdentityDictionaries are implemented as two parallel arrays. The first holds the keys, the second the values (Figure 2).

This implementation saves space because each association in a Dictionary takes 12 bytes of header + 8 bytes of object reference = 20 bytes. The total memory usage for a Dictionary is 12 bytes for the header of the Dictionary + 4 bytes times the basic size of the Dictionary + 20 bytes times the number of entries. The memory required for an IdentityDictionary is 24 bytes for the header of the object and the value collection + 8 bytes times the basic size.

For example, a 10,000-element Dictionary that has 5,000 entries free would take 12 + (4 * 15000) + (20 * 10000) = 260,012 bytes. You can see how the overhead of the Associations adds up. The same collection stored as an IdentityDictionary would take 24 + (8 * 15000) = 120,024 bytes.

OrderedCollection

OrderedCollections are like Arrays in that their keys are consecutive integers. Unlike Arrays, they are dynamically sized. They respond to add: and remove:. OrderedCollections preserve the order in which elements are added. You can also send them addFirst:, addLast:, removeFirst, and removeLast.

Using these methods, it is possible to implement stacks and queues trivially. There are no Stack or Queue objects in Smalltalk because it is so easy to get their functionality with an OrderedCollection. To get a stack you use addLast: for push, last for top, and removeLast for pop (you could also operate the stack off the front of the OrderedCollection). To implement a queue you use addFirst: for add and removeLast for remove.

As an example of using an OrderedCollection for a queue, let's look at implementing level-order traversal. Given a tree of objects, we want to process all the nodes at one level before we move on to the next:

```
Tree>>levelOrderDo: aBlock
    | queue |
    queue := OrderedCollection with: self.
    [queue isEmpty] whileFalse:
        [| node |
        node := queue removeFirst.
        aBlock value: node.
        queue addAllLast: node children]
```

OrderedCollections keep around extra storage at the beginning and end of their indexable parts to make it possible to add and remove elements without having to change size (Figure 3).

Because OrderedCollections are dynamically sized they preallocate a number of slots when they are created in preparation for objects being added. If you are using lots of OrderedCollections and most are smaller than the

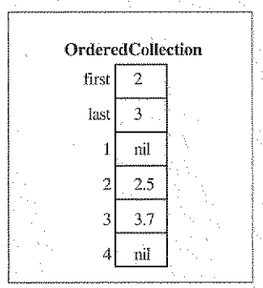


Figure 3. The result of (OrderedCollection new: 4) add: 2.5; add: 3.7.

GETTING REAL

Juanita Ewing

How to manage source without tools

Many Smalltalk programmers develop significant applications without any source-management tools. Although it takes a certain amount of discipline, small- to medium-sized applications can be developed without additional tools. This column will describe several sound practices for the successful management of application source.

The code in this column is for versions of Smalltalk/V under Windows and OS/2. The ideas are applicable to other versions of Smalltalk/V and to Objectworks\Smalltalk.

CONCEPTS

One concept is critical for successful management of application source:

- Never view your image as a permanent entity.

And there are two corollaries:

- Don't depend on your image as the only form of your application.
- Store your application in source form and rebuild your image frequently.

Viewing the image as a non-permanent entity doesn't necessarily imply that vendors are selling unreliable software. There are several ways an image can become non-functional, other than a serious Smalltalk bug or disk crash.

An image can become unusable because of some simple mistake on the part of a developer, such as accidentally removing a class that is relevant to the application under development. If the image is the only form of an application, recovering sources for an application class can be difficult and tedious. Another common mistake is the accidental deletion of the change log or changes file. The source for all the changes you've made to an image is stored in this file.

Not all motivation for storing an application outside an image derives from mistakes. When your vendor releases a new version, migration to the new version may be necessary to take advantage of new features or continue to the highest level of technical support.

PRACTICE

What is your application? In Smalltalk, this is not always a straightforward answer. Images contain large class libraries, and applications are developed by adding to and modifying class libraries. There is no clear distinction between system and application code. Because of this, it is very difficult to extract all parts of an application from an image, especially after the development is completed. It is better to extract or list the parts of your application as you develop it. Then short-term memory can help you decide if the modification you made was necessary for your application or if a temporary modification was needed for debugging. One of the most common errors is to omit a critical piece of one's application.

I will discuss two techniques for extracting your application code as you develop it. The first technique uses the browser to file out code right after it is developed. Most application code will be located in new classes, which can be filed out as a unit. Other application components are extensions to system classes, which can be filed out at the method level. The result of this technique is many small files.

There are dependencies among the classes defined in these files. For example, a subclass depends on its superclass. I use a script to reassemble all these files in correct order, rather than try to remember what the dependencies are. It is possible to create the script for reassembly at the same time the parts of an application are filed out.

Figure 1 contains a script for installing multiple files. The script consists of a list of file names, which is enumerated to install each file into the image.

```
"Read and file-in application files."
#(
'ExtendedListPane.cls'
'AviationGraphPane.cls'
'JetEngine.cls'
```

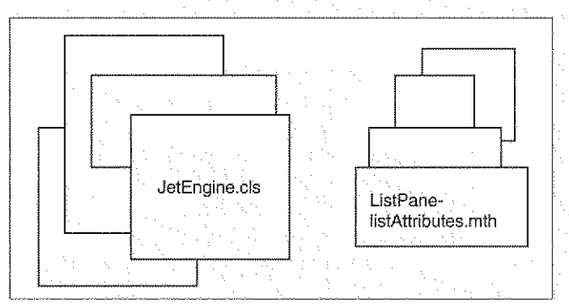


Figure 1. Example of reconstructing an application using multiple files.

```
'PropEngine.cls'
'RudderMechanics.cls'
'ListPane-class-supportedEvents.mth'
'ListPane-listAttributes.mth'
'ListPane-listAttributes:.mth'
'GraphicsMedium-bezierCurve:.mth'
)
    do:
        [:fileName |
        (Disk file: fileName) fileIn]
```

Another technique is to make a list of all relevant application pieces as they are developed. The list can be maintained in order of reassembly and used to extract all components of an application on demand. The result of extraction is a single file. Reconstruction of the application is a simple matter of installing one file. The source can be partitioned into several files, if necessary.

In Listing 1, the script has three lists: one for classes, one for instance methods, and one for class methods. The classes listed in the first script are written to the stream, then the methods in the second list are written to the stream. The file-out code makes use of ClassReader, which knows about Smalltalk source-file format.

This script makes use of a new method, fileOutClassOn:, defined in Listing 2. The new method, which writes a class definition and its methods on a stream, takes an instance of fileStream as an argument. It is similar to an existing method, fileOut:, which takes a file name as an argument, creates the file, then writes a class and its methods to the file:

The script in Listing 1 works in the simplest cases, in which there are no forward references to classes. For example, if code in the class JetEngine refers to the class PropEngine, the filein will not proceed properly. This problem can be avoided by defining all classes before any methods, as in the script in Listing 3. This script also has two lists, but the first list is enumerated over twice. A supporting method is defined in Listing 4.

Listing 1. Example of creating a single file for application reconstruction.

```
| sourceStream reader |
"Create filestream for storing sources."
sourceStream := Disk file: 'AviationSource.st'.
"Write application classes."
#(
ExtendedListPane
AviationGraphPane
JetEngine
PropEngine
RudderMechanics)
    do:
        [:className |
        reader :=ClassReader forClass: (Smalltalk at: className).
        reader fileOutClassOn: sourceStream].

"Write standalone instance methods"
#(
(ListPane listAttributes)
(ListPane listAttributes:)
(GraphicsMedium bezierCurve:)
)
    do:
        [:classNameAndSelector |
        reader :=ClassReader forClass: (Smalltalk at:
            (classNameAndSelector at: 1)).
        reader
            fileOutMethod: (classNameAndSelector at: 2)
            on:sourceStream].

"Write standalone class methods"
#(
(ListPane supportedEvents)
)
    do:
        [:classNameAndSelector |
        reader :=ClassReader forClass: (Smalltalk at:
            (classNameAndSelector at: 1)class).
        reader
            fileOutMethod: (classNameAndSelector at: 2)
            on: sourceStream].
sourceStream close.
```

INITIALIZATION

Applications consist of more than classes and methods. Instances of windows, panes, and domain-specific classes are also part of an application. Application reconstruction, therefore, must consist of more than filing in class and methods. The expressions executed in a workspace or inspector to set up the state of your application, such as initializing classes and creating new objects, need to be re-executed when your application is reconstructed. Save these expressions by collecting them in a file and executing them after reconstructing your application. In a future column I will discuss these types of expressions, and ways to execute them as part of a script.

ERRORS

The most error-prone portion of these techniques is recording pieces of the application as it is developed. Source-management tools are quite valuable because they record this information automatically. Because pieces of the application are recorded by hand, it is also common practice to search back through the change log to make sure no pieces have been for-

Listing 2. Supporting code in ClassReader for filing out a class onto a stream.

```
ClassReader
instance method

fileOutClassOn: aFileStream
    "Write the source for the class (including the class definition,
    instance methods, and class methods) in chunk file format
    to afileStream."
    class isNil ifTrue: [^self].
    CursorManager execute change.
    afileStream lineDelimiter: Cr.
    class fileOutOn: afileStream.
    afileStream nextChunkPut: String new.
    (ClassReader forClass: class class) fileOutOn: afileStream.
    self fileOutOn: afileStream.
    CursorManager normal change
```

Listing 3. Example of creating a single filefor application reconstruction.

```
| sourceStream classListreader |
"Create file stream for storing sources."
sourceStream := Disk file:'AviationSource.st'.
```

continued on next page

SMALLTALK IDIOMS

Kent Beck

Collections idioms: standard classes

Our previous column focused on enumeration methods and how to use all of them to advantage. This column covers the common collection classes, how they are implemented, when you should use them, and when you should be careful.

COLLECTION CLASSES

Array

Use an Array if you know the size of the collection when you create it, and if the indices into the elements (the first argument to at: and at:put:) are consecutive integers between one and the size of the array.

Arrays are implemented using the "indexable" part of objects. Recall that you can declare a class indexable. You can send new: anInteger to an indexable class and you will receive an instance with anInteger-indexable instance variables. The indexable variables are accessible through at: and at:put:. Array needs no more than the implementation of at: and at:put: in Object, and the implementation of new: in Class to operate.

Many people use OrderedCollections everywhere they need a collection. If you

- want a dynamically sized collection without the OrderedCollection overhead (see below)
- are willing to make the referencing object a little less flexible
- don't often add or remove items, compared with how often you access the collection

you can use arrays instead. Where you had:

```
initialize
    collection := OrderedCollection new
```

you have:

```
initialize
    collection := Array new "or even #()"
```

then you replace add: and remove: sent to collection with copyWith: and copyWithout: and reassign collection

```
foo
    collection add: #bar
```

becomes

```
foo
    collection := collection copyWith: #bar
```

The disadvantage of this approach is that the referencing object now has built into it the knowledge that its collection isn't resizable. Your object has, in effect, accepted some of the collection's responsibility.

ByteArray

ByteArrays store integers between 0 and 255 inclusive. If all the objects you need to store in an Array are in this range, you can save space by using a ByteArray. Whereas Arrays use 32-bit slots (i.e., soon-to-be-obsolete 32-bit processors) to store object references, ByteArrays only use 8 bits.

Besides the space savings, using ByteArrays can also make garbage collection faster. Byte-indexable objects (of which ByteArrays are one) are marked as not having any object references. The collector does not need to traverse them to determine which objects are still reachable.

As I mentioned in the last column, any class can be declared indexable. Instances are then allowed to have instance variables that are accessed by number (through at: and at:put:) rather than by name. Similarly, you can declare classes to be byte indexable. at: and at:put: for byte-indexable objects retrieve and store one-byte integers instead of arbitrary objects. A significant limitation of byte-indexable objects is that they can't have any named instance variables. This is to preserve the garbage-collector simplification mentioned above.

If you want to create an object that is bit-pattern oriented, but shouldn't respond to the whole range of collection messages, you should create a byte-indexable class. Such objects are particularly useful when passed to other languages because the bits used to encode the objects in a byte indexable object are the same as those used by, for instance, C, whereas a full-fledged SmallInteger has a different format than a C int.

Dictionary

Dictionaries are like dynamically sized arrays where the indices are not constrained to be consecutive integers. Dictionaries use hashing tables with linear probing to store and look up their elements (Figure 1). The key is sent "hash" and the answer modulo the basic size of the Dictionary is used to begin searching for the key. The elements are stored as Associations.

Dictionaries are rather schizophrenic. They can't decide whether they are arrays with arbitrary indices or unordered collections of associations with the accessing methods at: and at:put:. It doesn't help that Dictionary subclasses Set to inherit the implementation of hashed lookup. I treat them like arrays. If I want to

This claim provoked discussion about how easily register windows could be used—whether they would interfere with garbage collection (since values in registers outside the current window would not be easily visible) and other such topics.

Urs Hoelzle (urs@xenon.stanford.edu) mentioned that Self has been using SPARC register windows with garbage collection for some time. Peter Deutsch provided a comprehensive analysis of reasons for Smalltalk not to use them:

> The problem of pointers buried in register windows is indeed a significant one, but it is not the reason why I would recommend against modifying the Objectworks/Smalltalk (Ow/ST) implementation to use register windows. First, the performance gains would not be dramatic. Ow/ST spends a substantial fraction of its time in support code written in C, which would not be affected. A substantial fraction of the time in compiled Smalltalk code is spent doing message sends, type checks, etc., which would also not be affected. Also, since Smalltalk stacks get very deep and fluctuate more deeply than C stacks, the 7- or 8-register window on current SPARCs would over- and underflow significantly often. My best guess was that we would not see more than 20–25% performance improvement. (On future SPARC processors, where both the cost of memory references relative to register accesses and the number of register windows might be larger, this improvement might be somewhat greater.) Second, one of the keys to Ow/ST's remarkable portability is that it uses a very similar internal storage format for stack frames on all platforms. However, because saving and restoring register frames is done on the SPARC by code that is not accessible to ParcPlace, we cannot affect the storage format for these frames. So in order to use the SPARC register frames, we would have to either provide a complete second set of, or add radical new flexibility to, the large body of code in the runtime support system that manipulates stacks. The bottom line is that, in my opinion, the work required to fit Ow/ST to the SPARC's frame model would not justify the relatively small performance improvement. As for the comparison against Self, the Self authors acknowledge that the factor of 5 is only achievable under some circumstances. I do think it would be exciting to apply the Self compilation ideas to Smalltalk, and doing this could well produce dramatic performance improvements (on all platforms), but this would require wholesale redesign of most of the platform-independent code (other than the memory manager) in the Ow/ST runtime support system. The optimizing compilation experiments I did at ParcPlace were based on an alternative approach that would not have required such substantial changes to the Ow/ST virtual machine, but might have required type declarations (or at least type hints) provided by the user (or a type inference system). ■

Alan Knight is a researcher in the Department of Mechanical and Aerospace Engineering at Carleton University, Ottawa, Canada, K2C 3P3. He can be reached at +1 613.788.2600 x5783, or by e-mail at knight@mrco.carleton.ca.

...continued from page 5

month's system-dependent section because it depends on the layout of contexts.

Once the handler block is found, it's evaluated with the exception as a parameter. This allows the handler block to send the proceed, reject, restart, and return messages to the exception, and to query the exception for information about the error. Below are the implementations for proceed and reject—those for return and restart are in next month's article because they depend on some specifics of the V 286 system.

Proceeding is simple: Since we have the instance variable proceedBlock, all we need to do is evaluate it, perhaps with some meaningful parameter, as in

```
proceedDoing: aBlock
    "Return the value of aBlock as the value of the raise signal. Unwind
    the stack up to that point and resume execution in the context that
    raised the signal."
    | answer |
    answer := aBlock value.
    signalContext unwindLaterContexts.
    proceedBlock value: answer
```

Evaluating proceedBlock causes control to return into the context where the signal was first raised. The only subtle thing to remember concerns the unwind mechanism. Before evaluating proceedBlock, we call unwindLaterContexts, which evaluates the unwind blocks of every context we'll skip by proceeding.

Implementing reject is also quite simple. The current handler context (as found by fetchHandlerBlock:) is stored in the exception's handlerContext instance variable, so to find the next handler below the current one, we just need to look for some handler for the exception's signal below handlerContext. We can do that by sending propagatePrivateFrom: to the receiver exception with handlerContext as the parameter.

At this point we have a system-independent implementation for much of our package. The class Signal is complete and we need only three more methods for class Exception: return, restart, and the private method fetchHandlerBlock:. We also need to implement unwindLaterContexts to implement our unwind mechanism. Finally, we need some extra functionality for class Process. Next month, we will describe these final aspects of our system, such as the need to create a new set of context-related classes to make dealing with contexts in V 286 consistent and relatively trouble-free. ■

References

1. Van Orden, E. Application talk, HOOPSLA! 1(2), 1988.
2. Graver, J. Type-checking and type-inference for object-oriented programming languages. Doctoral thesis, University of Illinois at Urbana-Champaign, 1989.

Bob Hinkle and Ralph E. Johnson are affiliated with the University of Illinois at Urbana-Champaign. Mr. Hinkle's work is supported by a fellowship from the Fannie and John Hertz Foundation.

Listing 3 continued

```
"Classes in the application"
classList := #(
ExtendedListPane
AviationGraphPane
JetEngine
PropEngine
RudderMechanics).

"Write application class definitions."
classList
    do:
        [:className |
        reader :=ClassReader forClass: (Smalltalk at: className).
        readerfileOutClassDefinitionOn: sourceStream].

"Write the methods for the application class"
classList
    do:
        [:className |
        reader :=ClassReader forClass: (Smalltalk at: className).
        reader fileOutOn: sourceStream].

"Write standalone instance methods"
#(
(ListPane listAttributes)
(ListPane listAttributes:)
(GraphicsMedium bezierCurve:)
)
    do:
        [:classNameAndSelector |
        reader :=ClassReader forClass: (Smalltalk at:
            (classNameAndSelector at: 1)).
        reader
            fileOutMethod: (classNameAndSelector at: 2)
            on:sourceStream].

"Write standalone class methods"
#(
(ListPane supportedEvents)
)
    do:
        [:classNameAndSelector |
        reader :=ClassReader forClass: (Smalltalk at:
            (classNameAndSelector at: 1)class).
        reader
            fileOutMethod: (classNameAndSelector at: 2)
            on: sourceStream].

sourceStream close.
```

Listing 4. Supporting code in ClassReader for filing out a class definition without methods.

```
fileOutClassDefinitionOn:aFileStream
    "Write the source for the class (but not for the instance
    methods and class methods) in chunk file format
    to aFileStream."
    class isNil ifTrue: [^self].
    CursorManager execute change.
    aFileStream lineDelimiter: Cr.
    class FileOutOn: aFileStream.
    aFileStream nextChunkPut: String new.
    CursorManager normal change
```


gotten. This activity is usually performed in a regular fashion, such as before each snapshot.

Another common error is to rebuild an application on top of an image that has been used for development. This is not a good idea because the state of the image is unknown. There may be unwanted side effects from objects in the image. It is imperative, therefore, that the application is reconstructed from a clean, pristine image.

FREQUENCY

How often should the application be rebuilt? Early in development, when many classes are being created, the scripts are modified rapidly. It valuable to rebuild often to test the scripts; if they're too far out of sync with the application source, it can be difficult to debug the reconstruction process. In the middle stages of development the scripts are not in so much flux and the application doesn't need to be rebuilt so often to test them out. Other considerations may force application reconstruction, such as redesign of parts of an application. As the product is nearing completion, the development team may want to reconstruct the application often to confirm that the build process is bug-free. ■

Juanita Ewing is a senior staff member of Digitalk Professional Services. 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 class libraries for the first commercial-quality Smalltalk-80 system.

THE BEST OF comp.lang.smalltalk

Alan Knight

Smalltalk performance

Many people think of Smalltalk as slow. Unfortunately, they're right, especially as compared with the reference point of optimized C. This column will explore why Smalltalk code runs so slowly, just how slow it is, and the possibility for improvement.

WHY IS SMALLTALK SLOW?

Although surprisingly fast for what it does, Smalltalk is slow for various reasons. Conventional wisdom blames garbage collection. After all, Smalltalk collects garbage while those other, fast languages don't. Garbage collection does have a price, but not nearly as high as people think. More time-consuming is safety checking. Smalltalk checks all array references to make sure they are in bounds, every object reference for null values, every integer operation for overflow, and so on. C does none of these things.

If you have a compiler like Turbo Pascal, which allows you to turn array-bounds checking on and off, try doing it with a program that uses arrays. The effect on performance is very noticeable. I still leave checking on by default, and always turn it on when I'm trying to debug. When I learned C I wasted a lot of time trying to figure out how to turn on bounds checking, but I finally did. It involves paying a lot for an interpreter so my code can run more slowly than equivalent Smalltalk, but it's worth it.

Of course, these approaches have the advantage that you only pay the price during development. Safety features can be turned off when shipping the "bug-free" production code. It would be an interesting experiment for a vendor to provide a fast, unsafe version of the Smalltalk virtual machine for standalone applications.

Another important factor is message passing, for two reasons. First, message sends are a little pricier than function calls. You have to additionally figure out which function to call at runtime. However, the high cost of message sends is due to their number. Everything in Smalltalk except instance-variable access requires a message send. Even if messages cost less than function calls, the fact that there are so many more in the average Smalltalk program than the average C program makes Smalltalk slower.

HOW SLOW IS IT?

Quantitative performance measurements are always difficult. Results vary greatly between applications and minor changes can make a big performance difference.

Given this difficulty, we are fortunate to have someone with a good knowledge of the subject, at least with respect to ParcPlace Smalltalk. This impressive disclaimer is from Peter Deutsch (deutsch@smli.eng.sun.com)

> I was the principal designer and implementor of ParcPlace's Smalltalk code generators, including the portability architecture, the code generation framework, the stack management architecture, and the individual generators for 680x0, 80386, SPARC, MIPS, and RS/6000. The opinions expressed below are my own and should not be attributed to ParcPlace or to Sun.

He then writes:

> In my experience, based on a variety of both micro- and macro-experiments, the ParcPlace Smalltalk system does benchmark around a factor of 8 slower than optimized C for integer, structure, and array computation that does not contain large numbers of procedure-call-free loops. For straight-line integer computation, the ratio can get down as low as 4 or 5 to 1. (Of course, ParcPlace Smalltalk does overflow checking on all arithmetic operations, so any such comparison is not entirely appropriate.) For highly optimizable loops, especially ones involving access to arrays or strings (which ParcPlace Smalltalk always bounds-checks, and C never does), the ratio can get up as high as 40 or 50 to 1 under the most unfavourable circumstances, such as the 1-statement loops of strlen or strcpy.
> It is because of these things that ParcPlace recommends that, when necessary, users write their high-usage loops in C. Smalltalk's advantages are in areas other than highest performance for unchecked inner loops.

IS THIS FAST ENOUGH?

For many applications, this kind of speed is high enough. The numerous advantages of Smalltalk are worth the performance hit in these areas. For other application areas, the speed is definitely unacceptable, but this is partly psychological. If Smalltalk is running as fast as it reasonably can, we must either accept the performance or use another language. If, on the other hand, it runs slowly because the implementors haven't bothered to make it go faster, then we may get annoyed about it.

A strong voice for the possibility of improving performance comes from the implementors of Self. Self is a prototype-based language that is even more difficult to optimize than Smalltalk, but its implementation achieves much better performance. This is done using an extremely aggressive optimizing compiler. For example, Self exploits range information in integer computations. Using this information, it can omit overflow checks in cases where they're shown to be unnecessary.

Bruce Samuelson (bruce@ling.utafl.edu) doesn't think current Smalltalk performance is fast enough. He writes:

> ParcPlace, your dynamic compilation technology, is indeed impressive. . . . But you can do better, and you have chosen not to because you don't think it is high priority:
> 1) The Self authors claim in the literature that Smalltalk could be sped up by about a factor of 5. They claim in person that PPS is not interested in doing so (at least as of OOPSLA '91).
> 2) Mike Khaw's recent posting showed that Smalltalk did integer arithmetic in a tight loop about 1/8 the speed of C. . . . This is in the ballpark of what one would expect for such low level comparisons.
> 3) A Smalltalk "VM implementor" told me at OOPSLA '91 that the machine code generated by the dynamic translator is of "plain vanilla," unoptimized quality. For example, he thought the code for SPARC machines (he was not the SPARC VM implementor) did not make use of register windows, SPARC's idiomatic technique for passing function arguments efficiently. Perhaps he was wrong, or perhaps I misunderstood him, but times past when I've posted this and asked for comments from PPS, you have remained silent. It seems like this is one area in which you could apply some fairly standard optimization techniques in your VM that wouldn't require modifications to the compiler in the VI.
> 4) A PPS employee was engaged in a serious optimization project before he left PPS. I have not heard from PPS on the status of this project, except a comment I would paraphrase as follows: "We are impressed with the speed of forthcoming new machines [based, I suppose, on DEC Alpha, HP-PA, Intel 586, TI Viking, etc.] and feel that hardware vendors will solve possible Smalltalk performance problems."
> 5) Critique of (4): Yes, Smalltalk grows faster in proportion to the hardware. But so does every other language, and Smalltalk remains 5–10 times slower than C. The hardware vendors are not improving the competitive position of Smalltalk, except to make it feasible to use at all, and they already did that a few years ago. As machines get faster, applications get more ambitious and demand more cpu cycles. . . . A software vendor offering a development environment should regard decent optimization as a priority. Reviews of software products, whether of languages or applications, usually give performance a prominent place. You will make us, your customers, look better if you give us the tools to write blazing applications.
> 6) I have had to spend more time on optimizing my Smalltalk code than I would have liked, which has taken time away from more productive activities. I imagine this has happened to other programmers.
> 7) A turbocharged Smalltalk that could even modestly compete with C and C++ in speed would be an absolute dynamite product. How many of the postings to comp.lang.c++ give efficiency as a reason for using this "engineering compromise"? Take away efficiency as a criticism of Smalltalk and a lot of programmers and managers will take note.
> 8) Digitalk must have had some money to spend to able to buy out Instantiations. What if they put some of their money into doing a bang-up job at optimizing ST/V? Where would that leave ParcPlace?
> 9) Despite all these comments, which are directed to PPS in response to Tim Rowledge's posting, I realize that PPS is a small company with finite resources. Your founders have profoundly influenced the entire computer industry (GUIs, object orientedness) for the better. And you sell a very nice Smalltalk environment indeed. So I will counsel myself to remain patient and trust your marketing instincts. But please don't keep performance on the back burner forever. . .

REGISTER WINDOWS

There are quite a few complaints here, and I entirely agree with the main thrust that ParcPlace needs to place more emphasis on performance. I'd like to specifically deal with one of the claims that attracted particular attention on the net: the assertion that ParcPlace Smalltalk does not use register windows on the SPARC. For those of you even more blissfully ignorant of hardware than myself, I will attempt to explain register windows.

Machine registers are very fast to access and CPU designers like to have lots of them. The downside of this (apart from having to use valuable chip space) is that when there are many registers, more bits in the instruction word are needed to specify which one you want.

There are various ways of getting around this. One is to have more than one set of registers, used for different purposes (e.g., integer and floating point). The SPARC designers provided lots of registers, but made only a few of them visible at a time. By changing the register "window," you change which registers are visible.

Changing the window normally is done when making a procedure call. Rather than put arguments onto the stack, which is in main memory and therefore slow, one can put them into registers, then change the register window. Since the windows have some overlap, values put into the bottom of the register window of the calling routine will appear in the top of the window of the called routine. The arguments are immediately available and the called routine has its own set of registers to play with.

This technique can speed up procedure calls quite a bit. SUN claimed in some document I once read that register windows were aimed specifically at incrementally compiled languages like LISP and Smalltalk. In these languages, the compiler doesn't have as much time to think about how to optimize code and there are many procedure calls. Register windows are supposed to allow these calls to be easily optimized.

If SPARC can't or doesn't exploit SPARC register windows, it sounds like there's a serious communication problem between chip and language designers.

The Smalltalk Report

The International Newsletter for Smalltalk Programmers

February 1993 Volume 2 Number 5

MODULES: ENCAPSULATING BEHAVIOR IN SMALLTALK

By Nik Boyd

Contents:

This article proposes a new view of modules and how they may be added to the Smalltalk programming system. Modules provide a way to control the visibility of shared names. Modules also provide a way to hide the detailed collaborations among a group of Smalltalk classes organized as a subsystem. The organizing principles of classes and modules are orthogonal. Thus, modules also can be used to safely extend existing baseline classes.

The concept of a module and modular software development has existed for many years. A variety of programming systems has provided support for using separate name spaces to control the visibility of names used in a program. Examples include Modula-2[1] and Ada.[2]

Smalltalk systems use classes to encapsulate the structure and state of objects. Because Smalltalk classes can hide their internal state and serve as centers around which program behavior may be organized, they also may be considered modular. But while Smalltalk classes can encapsulate the state of their instances, they do not encapsulate their instances' behavior.

By convention, some messages are designated as "private" for the private use of the class and its instances. However, the Smalltalk system does not enforce designated message privacy and it is not always clear what such privacy means. For example, should subclasses be restricted from using private messages they inherit from their superclasses?

Because classes are globals in the Smalltalk system dictionary, they are all visible to all other classes. This visibility is excessive and it can contribute to information overload for novice Smalltalk programmers. It also can cause class naming conflicts when a team of developers integrate their separately developed components.

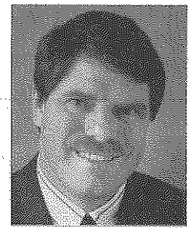
This article attempts to deal with these issues in a relatively nonintrusive manner that does not sacrifice any of the flexibility and power offered by existing Smalltalk systems.

MODULES

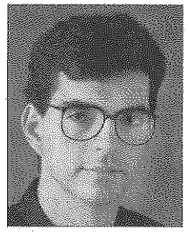
In their work on Modular Smalltalk,[3] Allen Wirfs-Brock and Brian Wilkerson describe the essential features of modules:

> Modules are program units that manage the visibility and accessibility of names . . . .
> A module typically groups a set of class definitions and objects to implement some service or abstraction. A module will frequently be the unit of division of responsibility within a programming team . . . .
> A module provides an independent naming environment that is separate from other modules within the program . . . .
> Modules support team engineering by providing isolated name spaces . . .

continued on page 4...



John Pugh



Paul White

EDITORS' CORNER

Most of you are probably satisfied with Smalltalk as a development tool. In fact, many of us feel even a bit arrogant about promoting Smalltalk as the "best" tool for developing software systems. Nevertheless, most would have to agree that we still lack an integrated process, and tools to match that process, for the entire software development lifecycle. Whether we're using CRC, Booch's notation, OMT, or something else, there still exists a "leap" from the design process to the construction of the software. Many of the tools on the market today offer little in the way of matching designs with the corresponding code.

Even more important, though, is the fact that we still don't have proper tools to allow us to go back and update the design to reflect changes in the construction. If we are to reap the benefits of the new "object-oriented lifecycle" many of us are advocating, where the design and development phases can be better integrated, we're going to need such tools. As Sam Adams and Steve Burbeck pointed out in the November/December 1992 issue of Object Magazine, "design is a continual process of discovering, evaluating, and deciding between alternatives." This can only be achieved if the costs of doing so are manageable.

One issue that has been addressed over the months in this newsletter is how best to manage visibility of objects within Smalltalk. On large development projects where teams work on subsystems to be integrated, managing the name space always proves to be a difficult task. In our feature article this month, Nik Boyd takes a new look at using modules as a vehicle for managing class library name spaces. He states that modules can be used to hide the details of the implementation of a software component consisting of a number of cooperating classes and he discusses means for implementing them.

In her column this month, Rebecca Wirfs-Brock calls for software development teams to characterize their objects. She states that such characterizations will help ensure that all team members are "in sync" and working toward a common system architecture. Through her experience, she proposes a number of terms for characterizing objects that could be adopted by your team.

Kent Beck offers two columns in one this month. As an aside, he describes a short idiom for testing nil values in an expression. The main column calls for software developers to "describe the intent behind a piece of code" for those who will later need to understand it to reuse it. Kent suggests that what is needed is a "pattern language" capable of describing these intentions at a variety of levels.

Juanita Ewing's "Getting Real" column addresses the problem common to all computing languages—how best to deal with defining and using both constants and default values. As she points out, to develop a code that will be reusable, default values must be defined in a consistent fashion and a mechanism must be provided for overriding them. In the "GUI" column this month, Greg Hendley and Eric Smith comment on the similarities and differences between Cooper and Peters' WindowBuilder and ParcPlace's new VisualWorks. The issues involved in copying Smalltalk objects arise regularly on USENET, and Alan Knight tackles some of these issues in his regular "comp.lang.smalltalk" column this month.

The Smalltalk Report

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

SIGS Publications
Advisory Board
Tom Atwood, Object Technology International
Grady Booch, Rational
George Bosworth, Digitalk
Brad Cox, Information Age Consulting
Chuck Duff, Symantec
Adele Goldberg, ParcPlace Systems
Tom Love, Consultant
Bertrand Meyer, ISE
Meilir Page-Jones, Wayland Systems
Sesha Pratap, CenterLine Software
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
Eric Smith, Knowledge Systems Corp.
Rebecca Wirfs-Brock, Digitalk

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

Art/Production
Kristina Joukhadar, Managing Editor
Susan Culligan, Pilgrim Road, Ltd., Creative Direction
Karen Tongish, Production Editor
Robert Stewart, Desktop System Coordinator

Circulation
Stephen W.Soule, Circulation Manager
Ken Mercado, Fulfillment Manager
John Schreiber, Circulation Assistant

Marketing/Advertising
Jason Weiskopf, Advertising Mgr—East Coast/Canada
Holly Meintzer, Advertising Mgr—West Coast/Europe
Helen Newling, 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 Baird, Conference Technical Manager
Amy Friedman, Projects Manager

Margherita R. Monck
General Manager

SIGS PUBLICATIONS

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

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.

Mailed First Class. Subscription rates 1 year, (9 issues) domestic, $65, Foreign and Canada, $90, Single copy price, $8.00. POSTMASTER: Send address changes and subscription orders to: The Smalltalk Report, Subscriber Services, Dept. SML, P.O. Box 3000, Denville, NJ 07834. Submit articles to the Editors at 91 Second Avenue, Ottawa, Ontario K1S 2H4, Canada.

inherent complexity. Object management needs to be integrated much more smoothly into the operating system services and made to fit naturally with object-oriented languages. In effect, you want the operating system support for objects to be as transparent as support for memory allocation and deallocation, file services, and so on. The approach must be sufficiently general that it can accommodate a range of languages, not just C++ and Pascal. There will always be a place for interpreted languages such as Smalltalk and Actor, and I hope that future object-oriented operating systems will make cross-language sharing of objects a reality.

Polymorphism unbound, Zack Urlocker, WINDOWS TECH JOURNAL, 10/92

OOP is inclusive, just as structured programming was two decades ago. It differs, however, from structured programming's traditional association with functional design methods such as functional decomposition, dataflow diagrams or data structure design. In OOP, objects are first categorized into classes and organized hierarchically according to their dependency and similarity. Each class comprises a set of attributes reflecting the objects' generally static properties and a set of routines (in Smalltalk, methods) that manipulate these attributes. Then relations between classes, such as inheritance, are designed . . .

Object-oriented computing, David C. Rine and Bharat Bhargava, COMPUTER, 10/92

..."In the object world you start by defining classes," explained Lanny Lampl, a technical consultant in Levi Strauss" Information Resources Group. "You have to parcel out the responsibilities of each object and decide how classes will interact with each other." Carrying out an object-oriented analysis turned out to be harder than switching to SmallTalk. "The syntax of the language is not the big thing," Lampl said. "The important thing is learning how to think about objects."

Levi Strauss cuts client/server pattern, Jean S. Bozman, COMPUTERWORLD, 11/16/92

PRODUCT ANNOUNCEMENTS

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

The Smalltalk Interface to Objective-C makes Objective-C objects look like Smalltalk objects. The interface is based on the simple concept that every remote Objective-C object can be represented by a local Smalltalk proxy object and every Objective-C class can be represented by a Smalltalk instance. Messages sent to a local Smalltalk proxy object are transparently forwarded to the actual Objective-C object it represents and the results are returned as Smalltalk objects. If the return value is an object ID, a proxy for that object is returned so that follow-on messages are also forwarded.

To the Smalltalk developer, there are just Smalltalk messages being sent to Smalltalk objects. To the Objective-C developer, there are just Objective-C messages being sent to Objective-C objects. The net result is that the two languages are very smoothly integrated. Developers no longer have to choose between using Objective-C or Smalltalk. They can use both languages together, each where it is best suited.

Berkeley Productivity Group, 35032 Maidstone Court, Newark, CA 94560, 510.795.6086, fax: 510.795.8077

The Object People Inc., a leading international provider of training, mentoring, and project development services in object-oriented technology, has expanded its educational facilities and launched a new internship program for Smalltalk programmers. The company specializes in the design and development of custom Smalltalk applications.

The new training facility allows the firm to offer an expanded schedule of open enrollment courses in Smalltalk/V, Objectworks\Smalltalk, and object-oriented concepts, analysis and design. In addition, the firm's Objectworks\Smalltalk courses now include the new VisualWorks application development environment recently introduced by ParcPlace. The Object People is also offering courses in PARTS, Digitalk's new "visual development tool" for OS/2.

The new internship program is designed to fast-track the development of accomplished Smalltalk programmers. Interns will have the opportunity to work on their own applications while having immediate access to assistance and guidance from experienced Smalltalk developers. Internships are flexible in duration and are spent at The Object People's educational facility in Ottawa. The program is available to both Smalltalk/V and Objectworks\Smalltalk developers. Participation in the program is strictly limited in view of the intensive one-on-one interaction required to make the program successful.

The Object People Inc., 509-885 Meadowlands Dr., Ottawa, Ontario, Canada, K2C 3N2, 613.230.6897, fax: 613.235.8256

Highlights

Excerpts from industry publications

DATABASES

. . . Is the decomposition of the Open OODB system into modules arbitrary, or will other efforts to build a system with similar functionality result in a similar factoring? It is too early to report that such experiments necessarily result in similar factorings, but the Open OODB's factoring into modules is very similar to the application integration framework being developed by the industrial consortium Object Management Group. . . . Thus, the OMG and the Open OODB architectures are almost isomorphic. It is interesting that one is viewed as an application integration framework architecture and the other as an OODB architecture. . .

Architecture of an open object-oriented database management system, David L. Wells, Jose/ A. Blakeley, and Craig W. Thompson, COMPUTER, 10/92

. . . The power of objects is in their robustness, extensibility, flexibility, and modularity. Actually I wish engineers did not have to know or care about objects. Except as interesting metaphors, they are not useful to any one but computer professionals. But we are not yet able to reach that level of information hiding. If you are selecting an engineering database management system today, it probably should be object-oriented—and if it isn't, you should know why not.

What's the big deal about objects?, Joel N. Orr, COMPUTER-AIDED ENGINEERING, 11/92

DESIGN

. . . Although it's nice that operating systems are becoming object-oriented for the user, there's no doubt that maintaining backward compatibility with a straight C API brings with it an

While providing many potential improvements to Smalltalk, the Modular Smalltalk system does not implement modules as first-class objects. Like many other programming systems, the Modular Smalltalk system uses modules only for organizational purposes. This article proposes a different view of modules as a special kind of Smalltalk class.

MODULES FOR SMALLTALK

The definition of a normal Smalltalk class includes a reference to a superclass, the name of the new subclass, and the names of any new instance and class variables added by the new subclass. Class variables are shared by all the instances of a class and are visible to all its methods and subclasses, if any.

In addition, the new subclass can provide its methods with access to named objects that are shared on a subscription basis. Certain names in the Smalltalk system dictionary are bound to global pool dictionaries that contain these sharable named objects. The new subclass can subscribe to these objects by including selected global names in its list of pool dictionaries. For example, a File class might be defined using the following message:

```
Object subclass: #File
    instanceVariableNames:
        'directory fileId name '
    classVariableNames:
        'PageSize '
    poolDictionaries:
        'CharacterConstants '!
```

Modules may be added to Smalltalk in a relatively straightforward manner. Details of how this can be done are presented in a later section. For now, we can say that each module is a class containing a name space, called its domain, instead of simply a pool of class variables.

There are several new messages for defining modules and the private classes contained in their domains. The definition of a module for managing an inventory might use the following message:

```
Object moduleSubclass: #InventoryManager
    instanceVariableNames: ' '
    classVariableNames: ' '
    poolDictionaries: ' '!
```

A new private class can be added to the domain of the InventoryManager class using the message:

```
Object subclass: #InventoryItem
    in: InventoryManager
    instanceVariableNames:
        'partNumber partName quantity '
    classVariableNames: ' '
    poolDictionaries: ' '!
```

In order to add a new private subclass of InventoryItem, we send the name of the private class (#InventoryItem) as a message to the InventoryManager module:

```
InventoryManager
InventoryItem subclass: #FloorItem
    instanceVariableNames:
        'storeLocation '
    classVariableNames: ' '
    poolDictionaries: ' '!
```

The issues involved in this breaking of the module encapsulation will be considered further in a later section.

Modules can be used to create nested subsystems. The following message creates a nested module for managing accounts in the InventoryManager module class:

```
Object moduleSubclass: #AccountManager
    in: InventoryManager
    instanceVariableNames: ' '
    classVariableNames: ' '
    poolDictionaries: ' '!
```

Figure 1 depicts the structural relationships between classes in the InventoryManager module. Note that the graphic design notation of OMT[4] has been extended slightly to show what classes are encapsulated inside a module class. The rounded rectangles represent module domains. Note that the Smalltalk system dictionary also is considered to be the system domain.

ENCAPSULATING PRIVATE BEHAVIOR

Modules provide three ways of encapsulating private behavior, all of which are based on their ability to encapsulate private classes:

- class groups (systems)
- baseline class extensions
- private methods

Each of these options will be discussed in the following sections.

PACKAGING OBJECT SYSTEM DESIGNS

One advantage of modules is that they provide a way for developers to package systems of components. During the design of a system of objects, groups of classes often know of each other

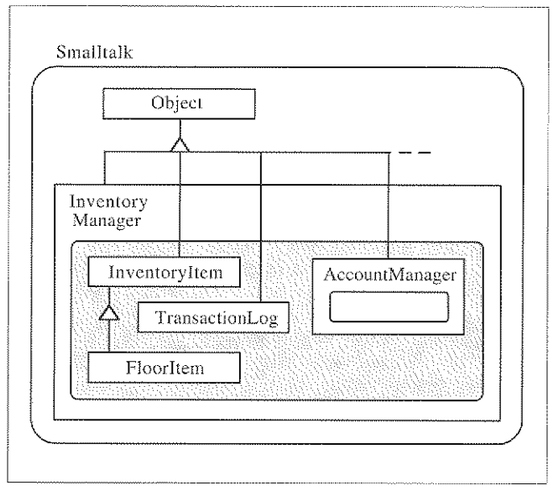


Figure 1. Structural relationships between classes.

the module grants direct access to an enclosed class by publishing it, then all the services of that class are directly available.

A module can provide direct access to an enclosed private class by supplying an accessing message as part of the public interface to the module. Suppose we want to give direct access to SubclassB in Figure 3. We could give ModuleA a class method named #SubclassB that answers SubclassB:

```
!ModuleA class methods !
SubclassB
    "Publish SubclassB."
    ^SubclassB! !
```

However, modules provide their greatest advantage when they hide or limit the visibility of their internals. This visibility is determined by what information (objects) is revealed by the module class and its instances (if any). The module forms the public interface to the classes inside the module domain.

> “Extending the visibility rules of the compiler is the key to adding modules to Smalltalk.”

COMPARISONS WITH OTHER WORK

Several other works [3,5,6] suggest that modules are not first-class and have no direct representation in an active system of objects. They suggest that modules only serve as name spaces for controlling the visibility of shared names. This article has presented a different viewpoint, advocating the inclusion of modules as a special kind of class.

Using a responsibility-driven approach,[5,7] the design of an object system can achieve a high degree of encapsulation and reusability. Classes help to maintain encapsulation when they limit access to their variables. Modules can help to maintain a higher degree of encapsulation by limiting access to the private behavior of subsystems.

The Law of Demeter[8] suggests that object systems can best realize the benefits of reuse by strictly limiting the visibility of objects to those other objects in the system that require such visibility. With classes and modules, visibility is controlled by the system designer.

CONCLUSION

This article shows how modules can be made first-class within Smalltalk systems. Modules provide a natural way of packaging object systems and give object system designers more options for controlling the visibility of a system's implementation details. Modules reduce the possibility of naming conflicts between separable systems of objects.

Just as classes form a hierarchy for the inheritance of structure and behavior, modules can be used to form a nested hierarchy of name spaces (domains). The organizing principles of classes and modules are orthogonal and complement each other.

Classes can be imported into modules by adding a private subclass of the same name to the module domain. However, given the new visibility rules for shared names, this kind of transparent subclassing may be the only reason for explicitly importing classes from outside a module.

Classes can be exported from a module by providing a message for accessing the class by name. However, this kind of revelation on the part of a module is discouraged because it leads to dependencies on the module's internals.

SOURCE CODE AVAILABILITY

Modules may be added to Smalltalk with relatively few changes. Two new classes and some changes to various core Smalltalk classes and the front end of the compiler provide the essentials for creating module classes. A tool for browsing module domains is included. This shows one way that support for modules may be integrated into the programming tools. The source code for adding modules to Smalltalk/V is available through the American Information Exchange (AMIX). ▣

References

1 Wirth, N. PROGRAMMING IN MODULA-2, TEXTS AND MONOGRAPHS IN COMPUTER SCIENCE, 2nd Edition, David Gries, Springer-Verlag, Berlin, 1984.
2 Booch, G. SOFTWARE ENGINEERING WITH ADA, Benjamin/Cummings, Menlo Park, CA, 1983.
3 Wirfs-Brock, A. and B. Wilkerson. An overview of modular Smalltalk. OOPSLA 1988 PROCEEDINGS, September 1988, pp. 123-134.
4 Rumbaugh, J. et al. OBJECT-ORIENTED MODELING AND DESIGN, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1991.
5 Wirfs-Brock, R. B. Wilkerson, and L. Wiener. DESIGNING OBJECT-ORIENTED SOFTWARE, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1990.
6 Szyperski, C.A. Import is not inheritance, why we need both: modules and classes. ECOOP 1992 PROCEEDINGS, June/July 1992, pp. 19-32.
7 Wirfs-Brock, R. and B. Wilkerson. Object-oriented design: a responsibility-driven approach. OOPSLA 1989 PROCEEDINGS, October 1989, pp. 71-75.
8 Lieberherr, K.L. and I. Holland. Formulations and benefits of the law of Demeter. SIGPLAN NOTICES, v24#3, March 1989, pp. 67-78.

Nik Boyd has been developing object systems since 1987. Since January 1990, he has been with Citicorp Transaction Technology Inc. in Santa Monica, California, where he is currently a Principal Member of the Technical Staff. His experience with OOP includes work with PARTS Workbench, Smalltalk/V for PM, Mac, Windows, and DOS, and Objectworks/Smalltalk v2.5 for DOS and v4.0 for Windows. His research interests include instance-centered and class-centered object systems, as well as tools and techniques that support object-oriented software engineering. Nik may be contacted via internet e-mail at 74170.2/71 @ CompuServe.com or through the American Information Exchange (AMIX).

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS MAIL PERMIT NO. 279 DENVILLE NJ

POSTAGE WILL BE PAID BY ADDRESSEE

The Smalltalk Report
Subscriber Services Dept SML
PO Box 3000
Denville NJ 07834-9821

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS MAIL PERMIT NO. 475 ALAMEDA, CA

POSTAGE WILL BE PAID BY THE ADDRESSEE

SERVIO CORPORATION
950 MARINA VILLAGE PARKWAY
SUITE 110
ALAMEDA, CA 94501

The Smalltalk Report

☞ **Provides objective & authoritative coverage on language advances, usage tips, project management advice, A&D techniques, and insightful applications.**

"If you're programming in Smalltalk, you should be reading **The Smalltalk Report"**

❑ **Yes, I would like to subscribe to The Smalltalk Report** Date_____

❑ 1 year (9 issues)
- ❑ Domestic $69.00
- ❑ Foreign $94.00

❑ 2 year (18 issues)
- ❑ Domestic $128.00
- ❑ Foreign $178.00

Method of Payment

❑ Check enclosed (payable to **The Smalltalk Report**)
❑ Bill me
❑ Charge my: ❑ Visa ❑ Mastercard ❑ Amex
Card No. _____
Exp. Date _____
Signature _____

Name _____
Title _____
Company _____
Address _____
City _____
State _____
Zip _____
Country _____
Phone _____

1. Which dialect of Smalltalk do you use:
- ❑ Smalltalk V
- ❑ Smalltalk-80
- ❑ Other _____

2. What is your involvement in software purchases for your department/firm:
- ❑ Recommend Need
- ❑ Specify Product
- ❑ Make Purchase
- ❑ None

3. Which operating system supports your software:
- ❑ UNIX
- ❑ DOS
- ❑ OS/2
- ❑ Windows
- ❑ Other _____

4. What is your company's primary business activity:
- ❑ Computer/Software Development.
- ❑ Manufacturing
- ❑ Financial Services
- ❑ Government/Military/Utility
- ❑ Educational/Consulting
- ❑ Other _____

5. For how long have you been using Smalltalk:
- ❑ Less than one year
- ❑ 1–3 years
- ❑ 3+ years

A member of the Object Marketing Network

fax to 212/274-0646

E3AG

SIGS PUBLICATIONS

Listing 1.

ClassFiler objects are responsbile for filing Smalltalk source code in and out of streams, usually FileStreams. This example is derived from the Smalltalk ClassReader. It shows how private methods can be encapsulated in a module.

```
"The public interface module class."
Object moduleSubclass: #ClassFiler
  instanceVariableNames:
    'privateSelf '
  classVariableNames: ''
  poolDictionaries: '' !

"The private ClassFiler class."
Object subclass: #ClassFiler in: ClassFiler
  instanceVariableNames:
    'class '
  classVariableNames: ''
  poolDictionaries: '' !

!ClassFiler class methods !
forClass: aClass
    "Answer a new instance of a public ClassFiler
     object."
    ^self new forClass: aClass! !

!ClassFiler methods !

fileInFrom: aStream
    "Read chunks from aStream. Compile each
     chunk as a method for the class described
     by the receiver. Log the source code of the
     method to the change log."
    | stream |
    stream := Sources at: 2.
    stream setToEnd.
    privateSelf instanceHeaderOn: stream.
    privateSelf fileInFrom: aStream.
    stream nextChunkPut: ''; flush!

fileOut: methodName On: aStream
    "File out the named method for the class
     described by the receiver to aStream, in
     chunk format."
    privateSelf checkFor: methodName.
    aStream cr.
    privateSelf instanceHeaderOn: aStream.
    privateSelf fileOut: methodName On: aStream.
    aStream nextChunkPut: ''; cr.!

fileOutOn: aStream
    "File out all the methods for the class described
     by the receiver to aStream, in chunk format."
    aStream cr.
    privateSelf instanceHeaderOn: aStream.
    privateSelf fileOutMethodsOn: aStream.
    aStream nextChunkPut: ''; cr!

forClass: aClass
    "Answer the receiver after attaching a new
     private instance of the private ClassFiler class."
    privateSelf := ClassFiler new setClass: aClass.! !

!ClassFiler ClassFiler class methods ! !

!ClassFiler ClassFiler methods !

checkFor: methodName
    "Verify that the class described by the receiver
     contains the named method."
    class methodDictionary
        at: methodName
        ifAbsent: [
            ^self error:
                methodName asString,
                ' is missing from ',
                class printString
        ].!

fileInFrom: aStream
    "Read chunks from aStream until an empty
     chunk (a single bang '!!') is found. Compile each
     chunk as a method for the class described
     by the receiver."
    | aString result |
    [   aString := aStream nextChunk.
        aString isEmpty
    ]
    whileFalse: [
        result := class compile: aString.
        result notNil ifTrue: [
            result value sourceString: aString
        ]
    ].!

fileOut: methodName On: aStream
    "File out the named method for the class
     described by the receiver on aStream, in
     chunk format."
    aStream cr; nextChunkPut: (
        class sourceCodeAt: methodName
    ).!

fileOutMethodsOn: aStream
    "File out all of the methods for the class
     described by the receiver on aStream."
    class selectors asSortedCollection do: [ :selector |
        self fileOut: selector On: aStream
    ].!

instanceHeaderOn: aStream
    "Write a header which identifies the class
     described by the receiver on aStream."

    "Note that filing in translates double bangs to
     single bangs and filing out translates single
     bangs into double bangs (like those used here)."

    aStream
        cr; nextPut: $!!;
        nextPutAll: class printString;
        space; nextPutAll: 'methods !!'!
```

Because these new visibility rules subsume existing rules, the semantics of normal classes continue to be supported.

BREAKING AND ENFORCING MODULE ENCAPSULATION

Because modules enclose and encapsulate their private classes, programming tools need a way to break the encapsulation of the module to create new classes inside the module. For this reason, a change has been made to class Class.

When a module class sends #doesNotUnderstand: aMessage, the message selector is checked to see if it is a capitalized unary selector that is the name of a private class inside the module. If so, the message answers the requested private class from the module. Otherwise, the message is dealt with using the existing #doesNotUnderstand: behavior.

This revised behavior is provided expressly for the compiler and development tools. This service breaks the encapsulation of the module similar to the way #instVarAt: breaks the encapsulation of an object.

To enforce the encapsulation of a finished module, the module can be closed by adding another version of #doesNotUnderstand: to the module class, overriding the one in class Class. This can be accomplished simply by sending the message #closeModule to the module class:

```
ModuleA closeModule.
```

This forces other classes outside the module scope to use the publicly defined interface to the module.

MODULE INTERFACES

The module that encloses a group of private classes can provide either direct or indirect access to the services of those classes. If

explicitly and collaborate closely to produce some complex behavior. Such subsystems are described informally in DESIGNING OBJECT-ORIENTED SOFTWARE[5]:

> Subsystems are groups of classes, or groups of classes and other subsystems, that collaborate among themselves to support a set of contracts. From outside the subsystem, the group of classes can be viewed as working closely together to provide a clearly delimited unit of functionality. From inside, subsystems reveal themselves to have complex structure. They consist of classes and subsystems that collaborate with each other to support distinct contracts that contribute to the overall behavior of the system . . . .
> Subsystems are identified by finding a group of classes, each of which fulfills different responsibilities, such that each collaborates closely with other classes in the group in order to cumulatively fulfill a greater responsibility . . . .
> There is no conceptual difference between the responsibilities of a class, a subsystem of classes, and even an entire application; it is simply a matter of scale, and the amount of richness and detail in your model...

This article goes beyond the conceptual to assert that there is no practical difference between the responsibilities of a class and a subsystem of classes when the subsystem is implemented as a module. The module class acts as a capsule around the subsystem of classes enclosed within the module domain.

Such packaging supports some of the practices of good software engineering. Implementation details can be localized, encapsulated, and scoped. Just as good object designs organize state and behavior into classes, systems of objects that are closely coupled, or that cooperate to provide some overall set of services, can be organized into modules.

EXAMPLE SYSTEMS

DESIGNING OBJECT-ORIENTED SOFTWARE gives several examples of object system design based on responsibilities, two of which are described in this article with just their class definitions. The first example already has been presented. The InventoryManager depicted in Figure 1 was derived from the Inventory subsystem described on pages 146–148 of the above book. Pages 151–152 describe the organization of a subsystem for managing transactions against financial accounts. Figure 2 shows how this subsystem might be organized as a module. The classes for this system could be defined using the following messages:

```
Object moduleSubclass: #FinancialManager
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''!
Object subclass: #Account
    in: FinancialManager
    instanceVariableNames:
        'accountID balance '
    classVariableNames: ''
    poolDictionaries: ''!
Object subclass: #Transaction
    in: FinancialManager
    instanceVariableNames:
        'account '
    classVariableNames: ''
    poolDictionaries: ''!
FinancialManager
Transaction subclass: #BalanceInquiry
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''!
FinancialManager
Transaction subclass: #FundsDeposit
    instanceVariableNames:
        'amount '
    classVariableNames: ''
    poolDictionaries: ''!
FinancialManager
Transaction subclass: #FundsWithdrawal
    instanceVariableNames:
        'amount '
    classVariableNames: ''
    poolDictionaries: ''!
FinancialManager
Transaction subclass: #FundsTransfer
    instanceVariableNames:
        'amount targetAccount '
    classVariableNames: ''
    poolDictionaries: ''!
```

EXTENDING BASELINE SMALLTALK CLASSES

Modules provide a safe way to extend and package changes to baseline classes in the Smalltalk system domain. Figure 3 shows how a private version of the String class can transparently subclass its baseline version so as to extend it.

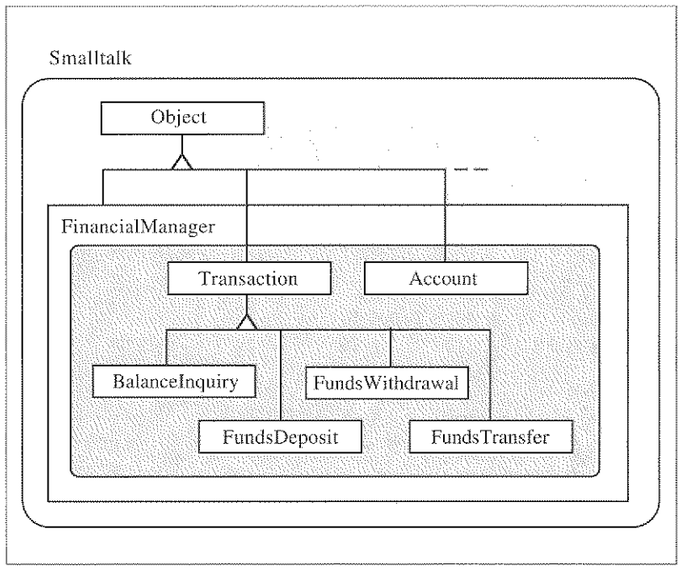


Figure 2. Subsystem organized as a module.

ModuleA is a moduleSubclass of class Object and SubclassB is a private class inside the domain of ModuleA. The private String class inside the domain of ModuleA is a private subclass of the baseline String class:

```
Object moduleSubclass: #ModuleA
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''!
    Object subclass: #SubclassB
        in: ModuleA
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''!
    String variable Byte Subclass: #String
        in: ModuleA
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''!
```

The private String class extensions are visible to methods in both ModuleA and SubclassB but not to classes outside of ModuleA in the Smalltalk system domain, such as SubclassC.

One drawback exists in the above example. The compiler creates constants for literals using the baseline classes: SmallInteger, Float, String, Symbol, and Array. Unlike Objectworks\Smalltalk, Smalltalk/V presently does not include the source code for its compiler. Because the Smalltalk/V compiler has not been extended to use the privatized versions of baseline classes it uses for literals, the private String class needs to create instances by copying baseline strings. For example, if we want SubclassB to use a private String for some operation, it will need to create it using:

```
"private" String copyFrom:
    'a constant string'
```

The compiler creates a constant string that is an instance of the baseline String class. The private String class creates an instance of itself that is a copy of this string constant. Given an instance of the private String class, the extended private string operations may be performed on it.

Given access to the source for the compiler, this small defect could be rectified. Then all the baseline classes, including those that the compiler uses for literals, could be extended transparently by private subclasses.

ENCAPSULATING PRIVATE METHODS

Modules can be used to hide the private methods of a class. To do this, a pair of classes is used to divide the public methods from the private ones. The public class is a module whose methods provide its public interface. The private methods are hidden in a private class inside the module domain. The private class can have the same name as the module class.

Figure 4 depicts an example of how this principle can be applied. Because of its simplicity, the full code for this example can be found in Listing 1. The module class ClassFiler is derived from the standard Smalltalk class ClassReader. This class is used to file Smalltalk source code in and out of the system, usually using an instance of class FileStream.

The ClassFiler module class has a single instance variable: privateSelf. When an instance of the module class is created, privateSelf is set to reference an instance of the private ClassFiler class. All the public methods of the module delegate private messages to privateSelf. Instances of the module class serve as proxies that hide the private behavior of the module class.

To maintain encapsulation, public methods in the module class can check the answers that come back from privateSelf. Any answer that is identical to privateSelf should be answered as self (the module instance) instead.

This technique provides true encapsulation of private methods of the class with a small amount of overhead in time (the delegation and answer checking) and space (the extra instance privateSelf).

continued on page 19

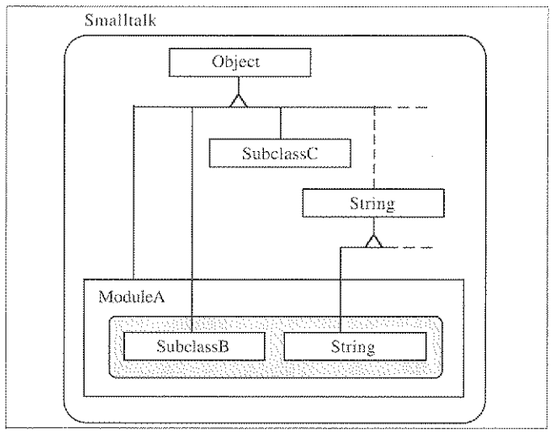


Figure 3. Extending a baseline class by transparently subclassing it.

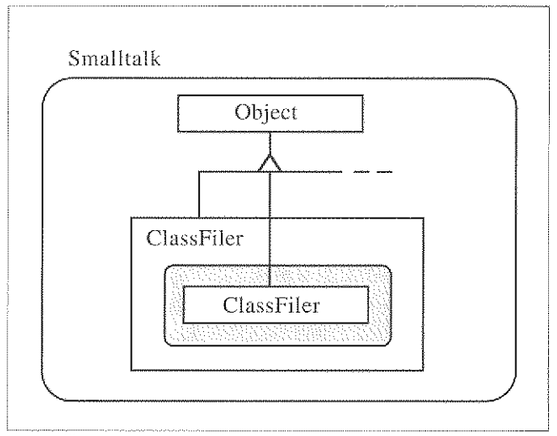


Figure 4. Using a module to hide the private methods of a class.

 continued from page 6

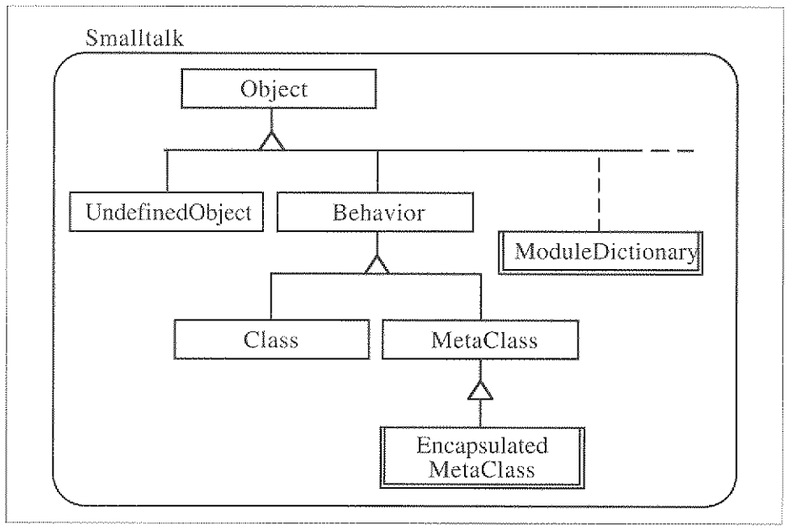


Figure 5.

ADDING MODULES TO SMALLTALK

Where a normal Smalltalk class uses a Dictionary for its pool of class variables, a module class uses a ModuleDictionary for its domain. The ModuleDictionary class is similar to the SystemDictionary class. Like the Smalltalk system dictionary, each module domain can contain shared objects, including other Smalltalk classes. In addition, each module domain keeps track of the names of the module class variables.

Each class contained in a module domain needs to know what module contains it. For this reason, each class contained inside a module domain is associated with an EncapsulatedMetaClass rather than a MetaClass. The class EncapsulatedMetaClass extends the class MetaClass by adding a reference to the module whose domain contains the encapsulated class.

Figure 5 depicts the classes changed to extend Smalltalk/V. Rectangles with doubled borders indicate the new classes.

RESOLVING SHARED NAMES

Smalltalk methods use names that start with lower case for private names, including instance variable names, method arguments, and block temporaries. Smalltalk methods also can reference shared objects whose names are capitalized.

The visibility of these shared names depends on where they are located in the system. Shared names can be found in class variable pools, global pool dictionaries, and the Smalltalk system dictionary. During method compilation, references to shared names are resolved by searching dictionaries in the following order:

- class variable pools of the class and its superclasses up through the class Object
- pool dictionaries to which the class subscribes from the Smalltalk system dictionary
- the Smalltalk system dictionary itself

Extending the visibility rules of the compiler is the key to adding modules to Smalltalk. The Smalltalk system dictionary is the enclosing domain for classes not contained in a module. As such, it is also considered the system domain. Because a module contains a name space in its domain, references to shared names are resolved by searching dictionaries in the following order:

- class variable pools of the class and its superclasses up through the class Object
- pool dictionaries to which the class subscribes in the module domains enclosing the class up through the Smalltalk system domain
- module domains enclosing the class up through the Smalltalk system domain

able to read a pattern and know :

- what problems need to be solved before this one can be solved
- what problem the pattern solves
- what constrains the solution to the problem
- what to do to the system to satisfy the pattern
- what problems to solve once this one has been solved

Patterns have a consistent structure. Each has the following sections:

- a name evoking the problem and its solution
- a prologue summarizing what other patterns have to be considered before this one is appropriate
- a one-paragraph preamble describing the crux of the problem solved by the pattern
- a diagram illustrating the problem
- a short essay exploring constraints on the solution
- one or two paragraphs describing how to solve the problem
- an illustration of the solution
- an epilogue summarizing patterns that can be considered once this one is satisfied

Several valuable traits are common to all patterns:

- They always call for concrete actions, even if they are at very high levels. For instance, a design-level pattern might call for splitting one object into two to improve flexibility. A coding pattern might help you give names to arguments.
- They include a complete description of the considerations influencing the solution. Almost no documentation describes the forces acting on a decision, but it is precisely this information that allows you to evaluate an object for usefulness in a particular context.
- They are illustrated with a simple diagram. Alexander's patterns are remarkable for the degree to which their essence can be distilled into a simple line drawing. The effective computer patterns I have discovered also boil down to a little picture.

The word "pattern" takes several meanings in this context. First, each solution represents a pattern of elements. The object that uses an OrderedCollection has a specific relationship with the objects it references. Second, the constraints acting on the solution form a pattern. The need to conserve space tugs this way, the desire for greater speed that way. Finally, and most curiously, are common patterns of human behavior. The act of choosing an OrderedCollection recurs many times and in many places.

PATTERN LANGUAGE

Patterns do not stand in isolation. The epilogue and prologue sections of each pattern link it to several others. The result can be seen as a kind of lattice, with problems that need to be addressed first higher than those that can be considered later. Much of an expert's skill comes from knowing what to worry about up front and what can be safely postponed. This process-oriented information is often as valuable as the patterns themselves.

The patterns together form a language in the sense that the patterns are terminal symbols, and the links between them are the productions. You create well-formed sentences by considering a sequence of patterns in turn. The result is a fully formed system. This is the primary difference between a pattern language and a set of design rules (like the Apple Human Interface Guidelines). The pattern language helps you create a system with the desired properties, not just analyze existing systems for the existence of those properties. A pattern language for good design will lead you to create a system with high coherence and low cohesion, not just describe the properties in isolation.

A complete pattern language for object-oriented programming encompasses patterns at all levels. Broad patterns cover issues like distribution of responsibility and control structures. Subsequent patterns help use the right abstractions in a library. Final patterns deal with variable naming, method naming, breaking methods into smaller methods, factoring code into inheritance hierarchies, and performance tuning.

CONCLUSION

No one has yet written a pattern language for objects like the one outlined above. There is general agreement that the problem of communicating intent is critical to cashing in on the promise of object-oriented programming. Researchers worldwide have turned to pattern languages as a promising approach to the problem. Here are a few I know about:

- Ralph Johnson at the University of Illinois is writing a pattern language for Hot Draw, a graphical editing framework.
- Richard Helm and John Vlissides of IBM and Erich Gamma of the Union Bank of Switzerland have been writing a catalog of "design patterns," which capture common design elements of C++ programs.
- Bruce Anderson of the University of Essex is leading an effort to compile an "architecture handbook."
- Oscar Nierstrasz at the University of Geneva has been using patterns to try to achieve reuse.

In subsequent columns I will explicitly use the pattern format where appropriate to describe Smalltalk idioms.I recommend the study of Christopher Alexander's work for those interested in attacking the educational side of the reuse problem. I have enjoyed studying the material both because of the obvious parallels between the pitfalls of professional architects and professional programmers, and because I am now far more sensitive to my physical environment Iand its effect on my life.

Architecture has the advantage (and disadvantage) of thousands of years of history to mine for patterns. Programming is a new enough discipline that we all have to invent new solutions often. Collecting and disseminating these common patterns will hasten the day we can get on to more interesting questions. As you discover patterns in your own work please send them to me. ▣

Kent Beck has been discovering Smalltalk idioms for eight years at Tektronix, Apple Computer, and MasPar 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, by phone at 408.338.4649, fax 408.338.3666, or compuserve 70761,1216.

PUTTING IT IN PERSPECTIVE

Rebecca Wirfs-Brock

Characterizing your objects

In this column I'll describe some vocabulary I find useful to characterize objects. Building an application involves teamwork and cooperation. Melding classes designed by individuals into a consistent system of cooperating objects requires that team members work toward a common system architecture. Team members need to share an understanding of what constitutes well-designed classes and subsystems, and what are acceptable patterns of object interactions.

Choices between perfectly acceptable alternatives must be made consistently across classes designed by different people. Achieving a consistent pattern of object communication first requires team members to use a common vocabulary for describing objects and their communication patterns. Once team members are talking the same language, they can have meaningful discussions about desirable interaction styles. Decisions then can be made based on sound engineering practices that meet business requirements.

STEREOTYPING OBJECT ROLES

Objects in our design can be either involved, active participants in many conversations, or by design play a more docile role, responding only when asked and taking a supporting role. Between these two extremes are many shades of behavior. I find it useful to classify objects according to their primary purpose as well as their modus operandi.

Here are two ways to characterize object roles:

- **Business Objects.** Objects whose primary purpose is to model necessary aspects of a concept that would be familiar to a user of the software we design. If we were designing an Automated Teller Machine for a bank, we might have Bank Customer, Bank Account, and Financial Transaction objects. If we were designing an oscilloscope we might model Triggers, Waveforms, or Timebases. These types of objects are also commonly referred to as domain objects because they correlate directly with concepts in the users' domain.
- **Utility Objects.** These are generally useful, non-application-specific objects. Smalltalk programming environments come with many generically useful classes. Classes for structuring other objects, such as Set, Array, Dictionary, and classes representing numbers or strings fall into this category.

There are compelling reasons for application developers to create additional utility objects. For several projects I've worked on, specific individuals were assigned direct responsibility for creating, publishing, and ensuring that utility objects were appropriate to the task and properly used. It is possible to create and effectively incorporate utility objects into the application throughout development and software construction.

It is extremely useful to design new utility objects that explicitly support system policies or common application programming practices. For example, we have created classes that stylize error handling and sequencing of processing steps; classes that model ranges of set table values, increments, and units of measurement; and classes that monitor detectable external conditions. Once designed, these objects can be used in many places within an application.

STEREOTYPING OBJECT BEHAVIORS

A number of researchers and design methodologists have coined terms for describing objects according to the way they operate. My list of useful terms isn't merely a composite of all common terms in the current literature. I continue to make finer distinctions after reflecting on past experiences and tackling new design projects. Periodic updating is needed to reflect new ways of constructing software that accomplishes new tasks.

Following are useful ways to classify object behavior.

Controlling objects

Controlling objects are responsible for controlling a cycle of action. This cycle can be either repetitive, with conditional branching logic, or initiated and executed once on detection of a certain set of events or circumstances. Controlling objects can initiate and control ongoing systemwide activity or iterate over a minor application task.

The original Smalltalk-80 user interface presented a stylized three-way collaboration between Model, View, and Controller objects. Controller objects were responsible for responding to user directives, such as mouse clicks or keystrokes, and initiating appropriate responses. Views displayed the current state of the application and model objects were application-specific objects.

I use a broader definition than that implied by Smalltalk-80 Controller objects. Controlling objects need not be spurred to action only on behalf of user directives. Controlling objects can be found and created for many parts of an application where a

cycle of activity is initiated, sequenced, and, sometime later, possibly completed.

For example, in the design of an Automated Teller Machine, an ATM object can have responsibility for initializing and sequencing system interactions with a bank customer. A further design refinement can add the concept of a Session-Controller object, which controls the sequence of activities by a single bank customer wishing to carry out one or more transactions with the bank. At a lower level, there may be network controller objects responsible for handling network traffic between the application and the communication network.

Coordinating objects

Coordinating objects are the traffic cops and managers within a system. Coordinators often pair client requests with desired services (or, rather, objects performing a requested service). In my early object design experience, I would append Manager to the names of these objects. FontManager and StyleManager are two example class names. I used to feel uncomfortable creating objects whose primary behavior was being idle until someone needed something, then helping to establish the connection between two other objects that would collaborate to actually perform some useful function. I now realize that these coordinators proved their worth simply by eliminating the need to hard-wire direct references between objects.

In another common design pattern, a coordinating object may respond to a request by briefly establishing an appropriate context, then delegating a request to one or more objects within its sphere of influence. For example, in the ATM design, the Session Manager first would determine which transaction the bank customer wished to perform, then create the appropriate transaction object for delegating the responsibility to gather additional information from the bank customer (such as amount to withdraw if it were a Withdraw Transaction), and then perform the transaction.

A coordinating object also may control a sequence of actions. It is often logical to blend coordinating and controlling functions in the same objects. A reasonable design for the Session Manager object is to give it the responsibility for creating and handling a series of bank customer transactions. A bank customer typically can perform transactions until indicating a desire to terminate the session, causing our application to print a receipt of all transactions and return the customer's card.

Structuring objects

Objects with structuring duties primarily maintain the relationships between application objects. In many applications, business objects have very complex structural relationships. Let's take a simplistic real-world example of a file cabinet containing folders that hold documents. A file cabinet simply holds folders that may be tabbed and labeled, and the folders merely contain their contents. The documents themselves are of interest.

In an object design, I add more or less behavior to objects to meet business requirements and to suit my personal tastes. I can design File Cabinets to do more than organize their contents. A File Cabinet could know when any folder was last referenced, or how much room is left in the cabinet. When I classify an object as primarily a structuring object, I think first and foremost about what relationships it should maintain between other objects and how it should do so, and secondarily what (if any) additional behavior might be appropriate and useful for it to have.

Informational objects

Sometimes objects are created to hold values that can be requested by many different kinds of application objects. I don't want to get into an in-depth discussion of design and programming techniques to eliminate globals or minimize dependencies on hard-wired values in code. However, at times it can be useful to create objects that are responsible for yielding information. In procedural programming languages, we have the ability to declare constant values. In object designs, informational objects are an equivalent concept.

Service objects

A service object typically is designed to perform a single operation or activity on demand. A well-designed service object provides a simple interface to a clearly defined operation; it should be easy to set up and use. Pure service objects often are the products of a highly factored design. Such a design consists of many classes of objects having highly specialized behaviors.

One reason to create service objects is to facilitate optional or configurable software features. The argument for this design strategy goes something like this: It is easier to configure a product's features by adding or removing entire classes of objects than it is to add or remove class behaviors.

As more behavior is added to a class, it can become complex to integrate new features with existing code. Optional functionality needs to be implemented in a way that guarantees pre-existing code doesn't break. Test suites and internal consistency checks become important.

When services are placed in specialized service classes, the design task shifts to creating an appropriate role and interface to the service object, which must balance the client's control over the service's performance with simplicity and ease of use.

An operation may be so complex to perform that it warrants creating many objects. A single object can be designed to provide the public interface to this service, hiding most of the details from the rest of the application.

Useful services can be packaged into distinct objects. These service objects might be designed so as to be useful in a variety of contexts, perhaps by being easy to extend or customize. We could design our ATM transaction objects to know precisely how to print information about the transaction on a receipt. Alternatively, we could design a Report object that provides printing and formatting services for the transaction object.

Interface objects

Interface objects are found at the boundaries of an object-oriented application. They can be designed to support communi-

SMALLTALK IDIOMS

A short introduction to pattern language

Kent Beck

This will be a departure from my code-oriented columns. For the last six months I've been surreptitiously presenting my material using a technique that I've been working with for the past six years or so. This technique was derived from work done in architecture (buildings, not chips) to help people design comfortable spaces for themselves. The time has come to tell you what I've been leading up to, so that I can directly refer to these concepts in the future.

First, though, I have to tell you about the most thoroughly useful little idiom I have seen in a long time. Ward Cunningham and I recently got to code together on a nifty spreadsheet project and he showed me a simple idiom for dealing with nil values. It saves me a line in many methods and, since most methods are three or four lines long, that's a significant savings. Here is the implementation:

```
Object>>ifNil: aBlock
    ^self

UndefinedObject>>ifNil: aBlock
    ^aBlock value
```

Simple, huh? Here what happens when you use it, though. You can transform code that looks like:

```
foo isNil ifTrue: [foo := self computeFoo].
^foo
```

into:

```
^foo ifNil: [foo := self computeFoo]
```

The savings comes because ifTrue: and ifFalse: return nil if the receiver is false or true, respectively. IfNil: returns the receiver, which can be any object, instead. I have found ifNil: useful in many more situations than the one listed above. Try it! If you find a clever use, send it to me and I'll write it up.

The one complaint about ifNil: is that it is slower than "isNil ifTrue:" (or its grosser cousin "== nil ifTrue:"). I claim that if you are focused on anything but achieving the most readable code possible in the middle 80% of a development, you're doing the wrong thing. Besides, it wouldn't be that hard to implement ifNil: as an inline message, just like the other conditionals. If it's not that hard, maybe I should write it up some time. Or maybe you should!

Now back to our regularly scheduled column...

The problem to be solved is describing the intent behind a piece of code to someone who needs to use it. There are plenty of methods for describing how code works (even though most programmers aren't disciplined in using them), but describing how code is supposed to be used is a black art. As the emphasis on programming shifts from just running programs to refining and reusing them, this is a problem of increasing importance. As objects are supposed to be about reuse, describing intent is of critical importance to us.

Donald Knuth has attacked the problem with what he calls "Literate Programming." He shares the insight that programmers ought to write programs for other programmers, not just the computer. His solution is to make programs read like books. When you read a literate program you are reading a combination of prose and code. You can filter out the non-program elements and run the result through a compiler to get an executable program.

There are a couple of problems with literate programming as Knuth conceives it. First, his literate programming system is implemented as a 1970s–style textual language. To write a literate program you have to know the programming language, the typesetting language, and the extensions required by the literate programming system. More importantly, the structure of a literate program is fundamentally linear. It is intended to be read from beginning to end. While this may be appropriate for a monolithic program like TeX, it does not address the problem of describing the intent of an object library, which is intended to be used piecemeal—sometimes just by instantiating objects, sometimes by plugging new objects into existing frameworks, and sometimes by refinement.

What we need is a structure for intention-oriented information that is flexible enough to convey a variety of information at different levels, but structured enough to provide a predictable experience for readers. It has to be able to convey process-oriented information but also describe programs piecemeal. It has to describe both how a program is intended to be used and how it works.

The solution I have been pursuing derives from the work of architect Christopher Alexander, who has spent many years seeking a way for architects to describe generic solutions to architectural problems so that individuals can adapt these solutions to their situations. The solution he found, called pattern language, solves all of the problems listed above: It is piecemeal, but also has large-scale structure; its essence describes the application of a solution, but also relates how the solution works; and it describes solutions at all scales, from urban planning to the size and color of trim in a house. His approach is presented in a pair of books from Oxford Press: THE TIMELESS WAY OF BUILDING and A PATTERN LANGUAGE.

PATTERNS

The unit of knowledge in a pattern language is a pattern. A pattern encodes an adequate solution to a problem known to arise in the process of building a system. A person should be

- provide buttons for committing the interface to code and for launching the interface
- use palettes that allow you to lay out panes or components as if you were using a drawing tool
- will open a browser on the generated code

One vs. many windows

The most obvious user interface difference between the two interface builders is the windows they use. WindowBuilder uses a single window with dialogs as needed. When a dialog is open the main window may not be used until that dialog is dismissed. VisualWorks makes use of a multitude of windows simultaneously, which some people call outboard windows. The window being built (the canvas) is in one window. and the outboard windows all operate on the canvas. Most, but not all, outboards operate on the most recently selected canvas.

Both techniques (outboards and dialogs) address the issue of clutter. The outboards allow users to decide how much information they want to see at once. However, this comes at a price. The canvas and the outboards are not visually tied together; it is not always clear which windows go together in VisualWorks, or even which windows are part of VisualWorks.

Resizing control

In both interface builders, the window being built responds to changes in the framing parameters of its panes or components. If a pane or component is given ratios instead of absolute positions, that pane changes shape as you change the initial size of the main window. WindowBuilder goes a step further and provides before and after silhouettes of your pane. As you change the framing parameters for a pane, it shows you a silhouette of your pane in the current window dimensions and also shows you the dimensions of your pane in a larger, resized window. This way WindowBuilder gives you immediate feedback.

SUMMARY

The two interface builders are more similar than different. The most important similarity is that they both fit nicely into the Interface part of the ICM framework, which lets you reuse design between dialects. After all, reuse of design is more powerful than reuse of code. ▩

Greg Hendley is a member of the technical staff at Knowledge Systems Corporation. His specialty is custom graphical user interfaces using various dialects of Smalltalk and various image generators. Eric Smith is also a member of the technical staff at Knowledge Systems Corporation. His specialty is custom graphical user interfaces using Smalltalk (various dialects) and C. The authors may be contacted at Knowledge Systems Corporation, 114 MacKenan Drive, Cary, NC 27511, or by phone, 919.481.4000.

■ GETTING REAL *continued from page 14*

```
    ^maxSize

initialMaxSize
    "Return the initial maximum size for text entry."
    ^32

initialize
    "Private - Initialize the receiver."
    value := ''.
    selection := 1@1.
    modified := false.
    ^super initialize
```

DEFAULTS REPLACE ARGUMENTS

Defaults also can be used to diminish interaction complexity. Commonly used values do not need to be passed as parameters; they can become defaults instead. Developers need to provide a way to override default values and still provide for the most common situations in which defaults are an applicable value.

The typical way for developers to provide default arguments is with additional methods that leave out key words. The method fill:rule:, from GraphicsTool, calls fill:rule:color: with the fill color set to the foreground color, which is a default. To override the default, the message fill:color:rule: can be sent:

```
fill: aRectangle rule: aRopConstant
    "Fill a Rectangle in the receiver medium with foreColor using
    aRopConstant."
    self fill: aRectangle rule: aRopConstant color: foreColor
```

CONCLUSION

The important difference between constants and defaults is their effect on reusability. Defaults, isolated in a method, are easily overridden by subclasses. Default values can be modified by instances if developers add enough support or can be used to eliminate arguments and reduce interaction complexity. Developers should always strive to evolve constants into defaults to make their classes more reusable. ▩

Juanita Ewing is a senior staff member of Digitalk Professional Services. 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 class libraries for the first commercial-quality Smalltalk-80 system. She can be reached at 503.242.0725.

cations with users, other programs, or externally available services. Interface objects come in many sizes, shapes, and flavors, and at many conceptual levels.

Interface objects can be designed to support an ongoing two-way communication between some external entity. For example, in the ATM application we have a number of physical devices such as Receipt Printer, Cash Dispenser, and Card Reader. In our design, all these devices would have interface objects that define a high-level interface to the services they provide. A Cash Dispenser object might define as message to dispense cash, return the cash balance, or adjust the balance (as a result of dispensing cash or adding more money to the machine).

Interface objects can be designed to translate external events or requests into messages fielded by interested application objects. For example, many external events need to be handled by the ATM system. To name a few: jamming of cash in the Cash Dispenser, failure of the door to close, the Receipt Printer running out of paper, etc. The list isn't endless, although responsible objects (the most likely candidates are appropriate interface objects) need to field those events and respond appropriately.

Or they can be designed to provide a narrow interface. For example, a menu presents a number of options and returns a user's preference. User interface objects typically support a highly stylized dialogue between the user and the system.

Interface objects are responsible for bridging the non-object world and the object world of messages and objects. When I think about interface object design, I focus first on those objects considered by the remaining applications to define the interface to the outside world. I realize that a great many details can and should be encapsulated by these interface objects. The key is to hide these details and provide a sufficiently abstract interface.

MOVING OBJECT DESIGNS ALONG THE BEHAVIORAL CONTINUUM

Given that we have a sufficiently rich vocabulary for describing object roles and behavioral patterns, we need to establish a context for applying these terms. Once we have done so, we need to evaluate our emerging design and select among alternatives. First it is useful to distinguish at what conceptual design level an object should belong (as opposed to where it is currently placed). Is it a high-level object or does it provide low-level services? Does it a play a significant or relatively insignificant role?

Once we determine this conceptual level, we can easily characterize an object's role as business or utility. Examining behaviors and building cleanly defined objects takes more time. Objects don't always fall into a single behavioral category, nor do I expect them to. For instance, objects often blend behaviors of controlling and coordinating. Another common pattern is to blend behaviors for structuring and providing services into the same object.

I do find it useful to ask whether an object is assuming too much responsibility, and whether it would be more appropriate to create new classes of objects to share the load. I also note whether a design choice causes an object's behavior to shift one way or the other on a behavioral continuum. Has an object become too active or passive? Is it perhaps taking on too many behaviors by assuming both a coordinating role as well as performing a useful service? Would it simplify the design to subdivide an object's responsibilities into smaller, simpler concepts? What would be an appropriate pattern of collaboration between that object and newly defined service objects?

When I look at rebalancing behaviors, I tend to consider the current behavior definitions for a group of collaborating objects belonging to roughly the same conceptual level. My goal is to understand and develop an appropriate distribution of control logic and responsibility among colloborators. Design creativity and individual preferences needn't be sacrificed during this assessment process. However, readjusting object behaviors needs to be purposefully done. In my next column I will discuss some object interaction styles as well as strategies and reasons for choosing between them. ▩

Rebecca Wirfs-Brock is Director of Object Technology Services at Digitalk and co-author of DESIGNING OBJECT-ORIENTED SOFTWARE. She has 17 years' experience designing, implementing, and managing software products, with the last eight years focused on object-oriented software. She managed the development of Tektronix Color Smalltalk and has been immersed in developing, teaching, and lecturing on object-oriented software. Comments, further insights or wild speculations are greatly appreciated by the author. She can be reached via email at rebecca@digitalk.com. Her U.S. mail address is Digitalk, 7585 S.W. Mohawk, Tualatin, OR, 97062.

THE BEST OF comp.lang.smalltalk

Alan Knight

Copying

Copying objects ought to be easy. After all, objects are just bits in the machine and those are easy enough to copy. Besides, objects are encapsulated, so copying shouldn't have to worry about anything outside the current object. Unfortunately, it's not always that simple. Complications can arise from details of Smalltalk's implementation and the object structure and from interactions with inheritance.

OBJECT IDENTITY

In Smalltalk, each object has a unique identity independent of the value it represents. In other words, Smalltalk variables don't hold objects but references to objects. Several different variables can refer to the same object; if a change is made to that object, the changed value is visible through all those variables.

This is also known as "aliasing" because the same object can have several different names, or "reference semantics" because the variables refer to the objects. This is in contrast to "copying semantics" where each variable has (or at least appears to have) its own copy of the object.

In pure functional languages, aliasing is eliminated. The values of instance variables in existing objects cannot be changed and new objects with different values must be created instead. Functional programmers would say that this is a good thing because it eliminates many confusing errors associated with aliasing. Non-functional programmers might say that removing aliasing entirely also eliminates many useful programming techniques but few would deny that copying semantics can be useful sometimes.

Some Smalltalk classes have copying semantics, including numbers, characters, booleans, and symbols. Operations on these types of objects do not modify the internal values of the instance but create a new instance as their result. Even though numbers can be aliased (as almost all Smalltalk objects can), there are no operations that can change the internal state and reveal the aliasing. The need to allocate new numbers for each operation results in poorer performance for numerically intensive applications but makes the behavior of numbers much more simple and predictable.

Complications

The previous section contains a number of half-truths. It's not really true that no operations modify classes with copying semantics. Meta-operations like become: and instVarAt: can get around these restrictions and it's possible to add methods that modify the internal state of some of these classes. In addition to seriously messing up your image, these facilities can expose significant differences in the behavior of these classes.

The most important difference, for copying purposes, is between SmallIntegers and all other objects. SmallIntegers are the most primitive entities in Smalltalk and really do have copying semantics, which the other classes just pretend to have.

The trick is that Smalltalk variables actually hold a 32-bit quantity, one bit of which is a flag. If the flag is set, the object referred to is a SmallInteger and the remaining 31 bits are its value. If the flag is not set, then it is some other kind of object and the remaining 31 bits are the machine address of that object.

If you copy the bits stored in a variable holding a SmallInteger, you actually get a copy of the SmallInteger. If the variable holds an object, then you get a copy of a reference to the object. This is the kind of implementation detail that you normally shouldn't have to think about, but it does explain a number of otherwise confusing things. For example, if you've ever wondered why become: doesn't work on SmallIntegers but does work on LargeIntegers, or why:

```
10 == 10
```

evaluates to true, but:

```
10 factorial == 10 factorial
```

comes out false, here is the explanation:

Become: can't work on SmallIntegers because it works by changing object references. SmallIntegers don't have object references, so there's nothing to be interchanged. In fact, since the parameter passing mechanism in Smalltalk is to copy these 32-bit fields described above, the become: operation doesn't even get the original SmallIntegers to change but only a copy of their values on the stack.

The == operation compares these same 32-bit quantities for equality. For SmallInteger 10, the bit patterns are exactly the same, so == is true. 10 factorial is a LargePositiveInteger, and since both sides of the expression are evaluated separately, we get two separate instances of LargePositiveInteger, which are equal (=) but not identical (==).

GUIs

Greg Hendley and Eric Smith

A quick look at two interface builders

In this installment of GUI Smalltalk, we will look at Smalltalk's two main interface builders: Cooper & Peters' WindowBuilder for different dialects of Smalltalk/V and ParcPlace's VisualWorks in R4.

While most people would not choose their Smalltalk dialect based on the interface builders available for it, it is interesting as a user and creator of graphical user interfaces (GUIs) to compare tools and see how two providers make use of GUIs themselves.

APPLES VS. APPLES OR ORANGES

The first question in comparing WindowBuilder and VisualWorks is "Are we comparing apples and apples or apples and oranges?" The answer is apples and apples. First, both are interface builders, not application builders; as such, their power is in graphically laying out the subpanes (if you are from V), controls (if you are from PM), or visualComponents (if you are from R4) of a window. This eliminates the need for you to calculate and write framing blocks.

COMPATIBILITY WITH THE ICM FRAMEWORK

Both WindowBuilder and VisualWorks output one class per window that can be used as the interface layer of the ICM framework. (The ICM framework was described in two previous installments of this column.) In WindowBuilder the default superclass of the output class is ViewManager. In VisualWorks the default superclass is ApplicationModel.

CAPABILITIES FOR CREATING USER INTERFACES

Similar capabilities

The capabilities of the two interface builders are more similar than different. Both have various versions of buttons, lists, static text, text editors, graphics, etc., and both help you build and test menus.

Sizing, positioning, and resizing of the window and its elements (subcomponents or subpanes) are supported in both. Elements of the user interface can be told to initially have the same width or height. They can be aligned like text: justified left, right, top, or bottom; centered horizontally; or centered vertically with respect to each other. Each element can be resized by absolute position or by ratios.

Both interface builders provide support for specifying each element's response to user input; both provide direct access to elements through the use of identifiers; both support tabbing; and, most important, both allow for the use of custom subpanes and visual components.

Differences in capabilities

Four capability differences between the two interface builders are noted below. Some are differences in degree while others appear in one but not the other; these include keyboard shortcuts, reuse, specifying response to user input, and specifying dependencies between components.

WindowBuilder provides direct support for keyboard shortcuts for menu items. VisualWorks does not provide such support from their tools.

VisualWorks provides support for three levels of user interface reuse. A user interface can be parameterized to work with any of a number of models. Inheritance can be used to let a subclass add visual components to its superclass. One interface can be used as a component in another interface. WindowBuilder only supports parameterization to use any number of models.

WindowBuilder provides support for specifying response to many types of user input. WindowBuilder tells you the events that may occur, lets you type the name of the method to invoke, and writes a stub for the method. For example, you can specify how to get the list for a list pane and what to do when a selection is made in the list pane. VisualWorks provides direct support only for specifying how to get the list. Responding to selection has to be explicitly coded in VisualWorks.

VisualWorks directly supports dependencies between different visual components in the same window. By making more than one component interested in a single aspect, all components respond when that aspect changes. Such dependencies have to be explicitly coded in WindowBuilder.

THEIR OWN USE OF USER INTERFACE TECHNIQUES

It is always interesting to see how the creators of an interface builder choose to use their tool. Let's start with their similarities.

Similarities

Both interface builders:

- operate in build-only mode

Instead, each constant should be defined in a separate method, allowing it to be easily identified and overridden. Once isolated, we call these values defaults because subclasses easily can override the defining method, increasing the reusability of the class.

The method initWindowSize, from the class WindowDialog, specifies the initial size of a dialog. Because this value is isolated in a method, we consider it a default—subclasses easily can override the default initial window size:

```
initWindowSize
    "Private-Answer the default window size."
    ^150 @ 100
```

Another example from the image involves the application framework class ViewManager. The class ViewManager has a method that specifies the class of the top pane in the view structure. Subclasses easily can override this method to specify another top pane class, giving subclasses the critical ability to override the creation of collaborators:

```
topPaneClass
    "Private-Answer the default top pane class."
    ^TopPane
```

EVOLVING CONSTANTS INTO DEFAULTS

In the section above, we saw two DiskBrowser methods containing an embedded constant, 10000. Next we see the two original methods rewritten, plus one other method that isolates the file size limit for automatic reading. The isolated constant is now a default because it easily can be overridden by subclasses. With a default, maintainers can locate the limit more easily and are less likely to create inconsistent methods caused by modifying one but not the other reference to the constant:

```
autoReadLimit
    "Return the file size limit that determines whether the entire contents
    of a file will be automatically displayed."
    ^10000

file: filePane
    "Private - Set the selected file to the selected one in filePane. Display
    the file contents in the text pane."
    | aFileStream |
    CursorManager execute change.
    self changed: #directorySort:.
    selectedFile := filePane selectedItem.
    self switchToFilePane.
    aFileStream := selectedDirectory fileReadOnly: selectedFile.
    wholeFileRequest := aFileStream size < self autoReadLimit.
    aFileStream close.
    wholeFileRequest
        ifTrue: [self fileContents: contentsPane]
        ifFalse: [self showPartialFile]

showPartialFile
    "Private - Display the head and tail of the selected file in the text
    pane."
    | aFileStream fileHead fileTail startMessage endMessagecr limit
    initial final |
    CursorManager execute change.
    limit := self autoReadLimit.
    initial := limit // 10 roundTo: 1000.
    final := limit - initial.
    contentsPane modified: false.
    aFileStream := selectedDirectory fileReadOnly: selectedFile.
    cr := String with: Cr with: Lf.
    startMessage :=
        'File size is greater than ', limit printString, ' bytes, ',cr,
        'first ', initial printString, ' bytes are ...', cr.
    endMessage :=
        cr, '**********', cr,
        'last ', final printString, ' bytes are ...', cr.
    fileHead := aFileStream copyFrom: 1 to: initial.
    fileTail := aFileStream
        copyFrom: aFileStream size - final
        to: aFileStream size.
    aFileStream close.
    contentsPane
        fileInFrom: (ReadStream on: (startMessage,fileHead,
        endMessage, fileTail));
        forceSelectionOntoDisplay.
    (self menuWindow menuTitled: '&Files')
    enableItem: #loadEntireFile.
    (self menuWindow menuTitled: '&File') disableItem: #accept.
    CursorManager normal change
```

INSTANCES MODIFY DEFAULTS

In addition to allowing subclasses to override defaults, developers can structure code so that instances can modify the default, improving client reuse. In this scenario, the class provides:

- storage for the default value, usually an instance variable
- accessing method for setting the default
- accessing method for retrieving the default (optional)

The class EntryField has a default for the maximum number of characters in an instance of EntryField. In addition to the initialize method we saw above and an instance variable to hold the value, one other method accesses the default maxSize. The accessing method maxSize: allows instances to customize the maximum number of characters that can be typed in an EntryField.

```
maxSize: anInteger
    "Set the maximum number of characters in the receiver to an Integer."
    maxSize := anInteger.
    handle = NullHandle
    ifFalse: [ self setTextLimit ]
```

There are several ways to provide an initial value for a default. In the initialize method for EntryField, maxSize is set to 32. An alternative design, shown below, has an accessing method that provides a default. The initialize method no longer sets the value of maxSize. In this case, the initial default value is only used if the default has not been otherwise set:

```
maxSize
    "Return the maximum number of characters that can be entered in
    the receiver. If no other value has been set, use the initial max size
    value and remember it."
    maxSize == nil
        ifTrue: [maxSize := self initialMaxSize].
```

continued on page 16

Shallow copy

How does this affect copying? The default copy implementation in Smalltalk is the "shallow copy," which just creates a new instance with exactly the same bits as in the old instance. This means we get a genuine copy of SmallIntegers and a shared reference to all other objects. Sometimes this is what you want but it also can be very confusing. For example, Richard Bentley (dik@comp.lancs.ac.uk) poses the frequently asked question:

> Could somebody please explain to me how copy is supposed to work. To me, if I take a copy of (say) a **Dictionary**, the copy should not just have pointers to the original **Dictionary**'s instance variables, so that if I change a value in my copy, the original is also changed.
> Is this how copy is supposed to work? If I want a **deepCopy** of a composite object (one that references other objects using instance variables), how should I go about it ?

Deep copy

In many cases, a deep copy is more intuitive than the one-level shallow copy. Deep copying has its own complications, though, and it's not possible to provide a single implementation that makes sense for all classes.

Digitalk provides an implementation of deepCopy that makes a copy of an object with shallow copies of all its instance variables. This is deeper than shallow copy but it just pushes the problem down one level. This wouldn't work properly in the dictionary example either because the instance variables of a dictionary are not the keys and values but the associations that hold them. They also provide an implementation of deepCopy specific to Dictionary, which does "the right thing." Such special implementations are required for quite a few classes, and still leave open questions like "How do I copy a dictionary of dictionaries?"

ParcPlace used to provide a recursive implementation of deepCopy, which would copy an object and make deep copies of all its instance variables, recursing until it reached primitive objects. This also has problems, as Bruce Samuelson (bruce@ling.uta.edu) points out:

> ParcPlace has been phasing out support for **deepCopy** because of theoretical problems such as infinite recursion for circular structures.

Jan Steinman (steinman@hasler.ascom.ch) adds:

> That's not good enough! **#deepCopy** has practical problems, such as chewing up memory when you least expect it. (Try to **deepCopy** a **SortedCollection**, for instance, which holds a **BlockClosure**, which holds a **CompiledLocalBlock**, which holds a metaclass, which links in the entire class tree. . . .)
> There are numerous solutions for avoiding infinite recursion, the simplest of which (context query) does not even require any additional state.
> I find **#deepCopy** so useful that I've implemented **#deepSize**, a "better BOSS," and lots of other deep things.
>
> They can be slow memory hogs, but if you use such things within their practical limitations, what's the problem? When **#deepCopy** goes away, I'll put it back!

The phrase "context query" hides a very clever trick that takes advantage of Smalltalk's reflective capabilities to avoid infinite recursion. Using the thisContext pseudo-variable in ParcPlace Smalltalk, it is possible to examine the stack of the currently executing process. This information can be used to determine whether an object already has been visited (and abort the recursion if it has). Jan Steinman has promised to write an article for THE SMALLTALK REPORT describing these tricks in detail. Similar tricks should be possible in Digitalk implementations but the interface to the process stack is not as well-documented, so it would take a bit more investigation.

Do it yourself

In general, if you want a copy routine that does "the right thing" for a particular class, you have little choice but to write it yourself. There isn't a universal definition of what the right thing is, and it may even vary for the same class from application to application. The problem of copying complex objects with circular references (e.g. a Graph) is equivalent to the problem of storing and retrieving an object from disk. In fact, if I have objects that can be written to a file, it's sometimes easiest to write them to a stream and retrieve them as a way of making a copy. There will be a big performance hit but sometimes that doesn't matter.

It's also worth noting that ParcPlace has changed default implementation of copy. Hans-Martin Mosner (hmm@heeg.de) writes:

> In R4.1, the only copy method besides **#shallowCopy** is **#copy** itself. It is implemented as **^self shallowCopy postCopy**. The **postCopy** method is the one that should do the dirty work. It can copy instance variables, leave others alone, update backpointers, and so on. Since it executes in the already copied object, it has access to everything it needs. To make copies which don't share instance variables, the **postCopy** methods should copy all such variables.

This is a nice implementation, since postCopy doesn't need to do anything for variables that only require a shallow copy. Thus, adding instance variables doesn't necessarily require changing the copy method. My only complaint is that this change was not very well advertised; I only discovered it by stumbling across the code while doing something else.

INHERITANCE

As if there weren't already enough problems with copying, there also can be problems inheriting from a class that defines its own copying methods. For example, Ralf Grohman (ralf@ubka.uni-karlsruhe.de) writes:

> I want to extend the **Dictionary Class** in some way. So I generated a new class (**Test**) which is a subclass of **Dictio-**

nary and added an instance variable **'temp'**.
The sole method of this class is:

```
addiere
 temp := 'ok'.
 1 to: 5 do: [:x |
Transcript show: 'temp='; show: temp printString; cr.
   self at: x put: 'Test'. ].
```

When I call it via **'Test new addiere.'** I get the following Transcript:

```
temp= 'ok'
temp= 'ok'
temp= nil
temp= nil
temp= nil
```

Hey! Why is the instance variable overwritten after the second iteration?

This problem is ParcPlace-specific and is described by Rick Klement (rick@rick.infoserv.com):

> It was not overwritten. It just wasn't moved to the new object created when the **Dictionary** had to grow to accommodate three entries. Welcome to one of Smalltalk's more subtle bugs. . . . I'll bet this bug exists in 10% of the large programs that add instance variables to variable classes.

ParcPlace Smalltalk implements classes such as Dictionary, Set, and OrderedCollections as variable classes (classes with indexed instance variables). When instances need to grow, a new, larger instance is created, and become: is used to replace the old collection with the new. Unfortunately, the grow method only copies the indexed instance variables. If non-indexed instance variables are present they must be copied explicitly, and user classes must override the grow method to do this. Jan Steinman (steinman@hasler.ascom.ch) writes:

> There have been many debates about how to best handle this. One might be an off-line "checker" method that would look for **SequenceableCollection** subclasses that add instance variables but do not implement **#grow.**
> I once reimplemented **#grow** so that it copied all instance variables, rather than specific ones (**1 to: self class instSize do: [:i | ...]**). But this gets you into trouble in some cases where the new **Collection** requires different values, such as **'firstIndex'** and **'lastIndex'** in **OrderedCollection** . . . . For the time being, the answer is to make sure people understand what is happening, but I've been Smalltalking for eight years, and it still bites me now and then!

Another possible solution is to implement collections differently. In Digitalk's version, these are normal classes that have an array as an instance variable. If the collection needs to grow, then a larger array is created, its contents are copied, and the instance variable replaced. It requires an extra layer of indirection for collection access, but become: is not necessary and the instance variables don't need to be copied. Digitalk's reason for doing this is probably that become: is a very expensive operation in their dialects, but Ralph Johnson (johnson@cs.uiuc.edu) argues that this is a cleaner implementation. In fact, he has code to change Smalltalk-80 to operate this way:

> I have a **fileIn** that will do this to 2.3, but haven't got around to doing it to any of the later images. You can't change classes like **MethodDictionary**, of course, but you can eliminate most of the old-style collections.

Alan Knight is a researcher in the Department of Mechanical and Aerospace Engineering at Carleton University, Ottawa, Canada, K1S 5B6. He currently works in ParcPlace Smalltalk on problems relating to finite element analysis and has worked in most Smalltalk dialects at one time or another. He can be reached at +1.613.788.2600 x5783.

GETTING REAL

Juanita Ewing

Constants, defaults and reusability

This column focuses on two aspects of reusability—subclassing and client usage—and how they relate to constants and defaults. Many classes have constants and defaults to represent commonly used values. Some of the values represented as constants may not really be constants, such as heuristically determined values, which are often hard-coded and embedded into methods. Though expedient in the prototyping stage, most constants should evolve into defaults as classes are refined. Developers of reusable software need to create reasonable defaults and include a mechanism to override them.

This column will show you how to use constants and defaults and still maintain a high level of reusability. We will examine several classes and methods from the Windows and OS/2 versions of Smalltalk/V that contain defaults. We will also revise some existing image code that has embedded constants and improve its reusability.

CONSTANTS

Many initialization methods contain constants and their values are often Smalltalk literals. In the class EntryField, the initialize method contains four constants: a string, an integer, a point, and a boolean. An initialization method is an appropriate place for constants. Subclasses typically override the initialize method to customize initial values:

```
initialize
    "Private - Initialize the receiver."
    value := ''.
    maxSize := 32.
    selection := 1@1.
    modified := false.
    ^super initialize
```

A less appropriate location for constants is embedded in arbitrary methods. A method should have one purpose: to define a default or perform some computation, but not both. With an embedded constant, reusability is impacted because it is difficult to:

- find and modify the constant
- override the constant in a subclass

The method file: in DiskBrowser has a constant that controls file contents display based on the file size. This constant is a size limit used to determine whether to display the entire file or a portion of it. If the file size exceeds this limit, it takes an extra action to see the entire contents. The main purpose of the file: method is to display the file contents. It should not contain the definition of the size limit:.

```
file: filePane
    "Private - Set the selected file to the selected one in filePane. Display
the file contents in the text pane."
    | aFileStream |
    CursorManager execute change.
    self changed: #directorySort:.
    selectedFile := filePane selectedItem.
    self switchToFilePane.
    aFileStream := selectedDirectory fileReadOnly: selectedFile.
    wholeFileRequest := aFileStream size < 10000.
    aFileStream close.
    wholeFileRequest
        ifTrue: [self fileContents: contentsPane]
        ifFalse: [self showPartialFile]
```

Another DiskBrowser method, showPartialFile, also contains this constant. Having the same embedded constant in two methods can lead to maintenance problems:

```
showPartialFile
    "Private - Display the head and tail of the selected file in the text
pane."
    | aFileStream fileHead fileTail startMessage endMessage cr |
    CursorManager execute change.
    contentsPane modified: false.
    aFileStream := selectedDirectory fileReadOnly: selectedFile.
    cr := String with: Cr with: Lf.
    startMessage := 'File size is greater than 10000 bytes, ',cr,
'first 1000 bytes are ...', cr.
    endMessage := cr, '**********', cr,
'last 9000 bytes are...', cr.
    fileHead := aFileStream copyFrom: 1 to: 1000.
    fileTail := aFileStream
        copyFrom: aFileStream size - 9000
        to: aFileStream size.
    aFileStream close.
    contentsPane
        fileInFrom: (ReadStream on: (startMessage,
fileHead,endMessage, fileTail));
        forceSelectionOntoDisplay.
    (self menuWindow menuTitled: '&Files') enableItem:
#loadEntireFile.
    (self menuWindow menuTitled: '&File') disableItem: #accept.
    CursorManager normal change
```

DEFAULTS

Developers should not embed constants in arbitrary methods.

The Smalltalk Report

The International Newsletter for Smalltalk Programmers

May 1993 Volume 2 Number 7

TOWARD A SMALLTALK STANDARD: TECHNICAL ASPECTS OF THE COMMON BASE

By R.J. DeNatale & Y.P. Shan

Contents:

Recognizing Smalltalk's increasing importance as a mainstream programming language and acting as a large user of the language, IBM recently proposed the formation of a standards effort within ANSI to define a Smalltalk language standard and offered a "common base" strawman to start such an effort. At this time the proposal has been accepted by the ANSI SPARC committee, and the formation of an ANSI Smalltalk committee has begun.

This article focuses on technical issues regarding the common base. We have written a companion article that will appear in OBJECT MAGAZINE, which outlines the history of the development of the common base.

WHAT IS THE COMMON BASE?

As part of the proposal for an ANSI Smalltalk standards effort, we have contributed a "strawman" as the starting point for standardization. That strawman is contained in the IBM document entitled Smalltalk Portability: A Common Base and comprises chapters 3–5 and appendices A and B from that document.*

This proposal is not our work entirely. It is the result of an 18-month-long collaboration among five companies: IBM, Digitalk, KSC, OTI, and ParcPlace.

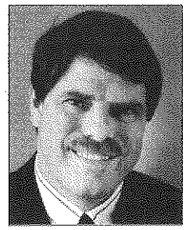
A purely syntactic description of Smalltalk results in a language specification that is incomplete when compared to those for languages such as C, COBOL, and FORTRAN. When studying the specification for a language one expects to learn things, such as how to do arithmetic, how to code conditional logic, and so forth. Smalltalk syntax does not address these issues. To bring the description of Smalltalk up to the expected degree of completeness we must specify a number of classes, such as numbers, booleans, blocks, and so on. The purpose of the common base is to provide a semantic description that is common to both Smalltalk-80 and Smalltalk/V. We wanted to produce a specification of Smalltalk that covers the variety of existing implementations. This led us to specifying the external behavior of classes without prescribing implementation. Detail differences between the two implementations were left out of the common base, although we have kept careful note of these differences in the review process, and they will no doubt be important items of discussion as the standardization effort proceeds.

Currently, the common base covers the following areas. (This scope might be changed during the standardization process):

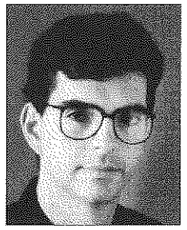
- Language syntax
- Common object behavior
- Common class behavior

continued on page 4...

** The document can be ordered from your local IBM branch office or by credit card through the IBM publications ordering number (800.879.2755). The publication number is GG24-3903. The price is $2.75 per copy for printing and handling.*



John Pugh



Paul White

EDITORS' CORNER

This month's hot topic is standards. After many years of discussion, an ANSI Smalltalk language standard is now much nearer to becoming a reality. There is little doubt that languages achieve an extra measure of respectability when an ANSI standard is defined for them. Many in the Smalltalk community have long recognized this, but how do you standardize Smalltalk? The language itself is very small, but standardization alone—though valuable—does not produce a very useful result. We must standardize the class library. The Smalltalk class library can be thought of as an extension of the language; for example, even control structures are captured via message passing rather than hard-wired syntactic constructs. However, now we run into further trouble. There are two major dialects of Smalltalk: Smalltalk-80 and Smalltalk/V. Enfin might be included as yet a third dialect, and by the time you read this article there may be a fourth, SmalltalkAgents for the Macintosh. Each has classes and frameworks unique to itself particularly in the domain of user interface classes. Even when we restrict ourselves to magnitudes and collections we are not out of the woods. Smalltalk-80 and Smalltalk/V have distinct differences both in the organization of the class hierarchy and in the classes themselves. How have all these issues been addressed? Well, read the lead article written by Rick DeNatale and Y.P. Shan and you will find out. For our part, we applaud the initiative taken by IBM to promote the standards effort and the participating vendors for putting their competitive instincts to one side for the benefit of the Smalltalk community as a whole. We'll keep you informed as the standardization effort proceeds and hope that as many of you as possible will play a part in the process.

But there's even more news on standards. Digitalk has announced that it will make its Smalltalk products interoperable with SOM, IBM's System Object Model for OS/2 2.0 and that it will also develop client-server database and development tools adhering to the data access portions of Apple's Virtually Integrated Technical Architecture Lifecycle (VITAL).

In our second feature article this month, Ralph Johnson provides us with a list of classic Smalltalk bugs. He has compiled his list from the collective experiences of many experienced Smalltalk programmers. The list will be particularly useful to beginning Smalltalk programmers. If you are aware of other bugs you think should be accorded "classic" status, please forward them to Ralph. His address is given at the end of the article.

In her column, Rebecca Wirfs-Brock passes on some more of her nine years of experience designing, implementing, and managing software projects. In this issue, she discusses the incremental nature of design and what distinguishes incremental design from rapid prototyping. In this issue's Getting Real column, Juanita Ewing discusses the inappropriate use of arrays and how their misuse affects reusability. Kent Beck continues his discussion on instance-specific behavior, where methods can be attached to individual instances, as opposed to being attached only to the class. This month, Kent explores the implementation of instance specialization in Digitalk's Smalltalk/V for OS/2 and contrasts it with the ParcPlace implementation of the same concept.

Finally, Alan Knight focuses on the thread of discussion generated on comp.lang.smalltalk by the following question: "I [have] always found a way to avoid this, but I would like to know how to break away from inside a loop and return [to] the immediate upper level context?"

We hope you enjoy this issue. John Pugh Paul White

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. Mailed First Class. Subscription rates 1 year, (9 issues) domestic, $65, Foreign and Canada, $90, Single copy price, $8.00. POSTMASTER: Send address changes and subscription orders to: The Smalltalk Report, Subscriber Services, Dept. SML, P.O. Box 3000, Denville, NJ 07834. Submit articles to the Editors at 91 Second Avenue, Ottawa, Ontario K1S 2H4, Canada. For service on current subscriptions call 800.783.4903. Printed in the United States.

■ SMALLTALK IDIOMS

...continued from page 17

tem, please pass them along. You'll find my address at the end of the article.

CONCLUSION

Instance specialization has a place in the toolbox of every experienced Smalltalker. You won't use it every day—maybe not even every year—but when you want it, nothing else will do. The implementations for VisualWorks and Smalltalk/V OS/2 2.0 are quite different, but they present the same external interface to the programmer.

The contrasts between the implementations hint at fundamental differences in approach between Digitalk engineering and ParcPlace engineering. I will explore the practical consequences of this difference in future columns. ▣

Kent Beck has been discovering Smalltalk idioms for eight years at Tektronix, Apple Computer, and MasPar 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, or at 408.338.4649 (phone), 408.338.3666 (fax), 70761,1216 (Compuserve).

■ THE BEST OF COMP.LANG.SMALLTALK

...continued from page 19

> In general, however, I think this technique is inferior to simply restructuring your code to have an inner method that can perform the loop and that can return from the loop when needed.

In the end, I have to agree that restructuring the code is usually the best solution. The number of different possibilities available does serve, however, as a reminder of the powerful facilities available in Smalltalk. ▣

ERRATA

Jon Hylands, an alert colleague who obviously reads my columns very carefully, has pointed out an error in a recent column on copying (February 1993). I had said that adding named instance variables to indexed collections in ParcPlace Smalltalk required overriding the grow method to copy these variables. In fact, the method that should be overridden is `copyEmpty`, which will be called by `grow`.

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

guage biased. Class libraries developed in one language cannot be used with other languages. For example, a class library developed in C++ cannot be used by a Smalltalk programmer, and a Smalltalk library is of no use to a COBOL programmer. The System Object Model (SOM) is a new packaging technology designed to address this and other packaging issues. . . .

In the current version of SOM as released on OS/2 2.0, we provide full tool support for only C language bindings. . . . We also have experimental C++ bindings, designs for Smalltalk bindings, and binding to an experimental object-oriented version of REXX.

Developing with IBM's System Object Model (SOM), Roger Sessions, First Class, OMG NEWSLETTER, Feb/Mar 1993

[Mel Beckman, Duke Communications Int'l.]: One brass-tack thing you can do to improve your professional perspective is to buy Smalltalk/V for the Mac or PC and go through the tutorial. In about a week of evenings, you will pick up more insight into object-oriented programming and where new design programming is headed than you will in two or three seminars. . .

[Nick Knowles, Steam Intellect, Ltd.]: We may be hearing about C++ from IBM Toronto, but we are also hearing about Smalltalk from Rochester. Smalltalk is probably a better fit for high-level business problems. C++ may give better performance for low-level tools. . .

[Paul Conte, Picante Software]: . . . What's important is to pick a language that lets you go through the exercise of building something with objet-oriented techniques. Then you'll see that while object-oriented languages may help solve some syntactic-level problems and code-organization problems, these languages lead to another generation of problems—the creation and management of class libraries. . .

Roundtable 1992: Change and challenge, Dale Agger, NEWS 3X/400, 12/92

...continued from page 1

- Magnitude
- Collections
- Streams
- Basic geometry
- File in/out format

THE TECHNICAL APPROACH

We wish the common base to describe the behavior of Smalltalk classes without prescribing implementation. To this end we have:

1. Documented only the public protocols of the classes
2. Avoided the specification of inheritance hierarchies

We will describe how we approached the specification of the collection classes without the prescription of a particular inheritance hierarchy.

COLLECTIONS

Collections are an important part of the Smalltalk class library, and present an interesting challenge given the desire to describe behavior without recourse to describing implementation inheritance.

A major inspiration for this work was the early publication on the internet by William Cook, currently with Apple, of his investigation of the relationship between the implementation and type hierarchies of the Smalltalk collection classes.[1] Following this work, we described each collection class individually without recourse to inheritance, in terms of combinations of the following protocols:

- **Expandable.** Contains the messages for adding elements to a collection. Set, SortedCollection, and OrderedCollection support this protocol.
- **Ordered.** Contains the messages that pertain to collections which maintain their contents in a specific order. SortedCollection, OrderedCollection, Interval, Array, and String support this protocol.
- **Copy-Replaceable.** Contains the #copyReplaceFrom:to:with: message. Interval, Array, OrderedCollection, and String support this protocol.
- **Array-Like.** Contains messages for changing the collection based on a collection or range of indices. Array, OrderedCollection, and String support this protocol.
- **Indexable.** Contains the #at: message used to access an element of the collection based on an index or key. SortedCollection, OrderedCollection, Interval, Array, String, Dictionary, and IdentityDictionary support this protocol.
- **Updatable.** Contains the #at:put: message used to replace an element of the collection based on an index or key. OrderedCollection, Array, String, Dictionary, and IdentityDictionary support this protocol.
- **Contractable.** Contains messages for removing an element or collection of elements from the collection. Set, SortedCollection, and OrderedCollection support this protocol.
- **Insertable-From-Ends.** Contains messages for adding elements at the beginning or end of the collection. OrderedCollection supports this protocol.
- **Removable-From-Ends.** Contains messages for removing elements from the beginning or end of the collection. SortedCollection and OrderedCollection support this protocol.

By specifying each collection class in terms of a set of these protocols we can describe the capabilities of each class without requiring a particular implementation hierarchy.

> “Smalltalk is more than ten years old. A standard is needed, and the time is now.”

FUTURE STANDARDS ACTIVITY

The common base represents an attempt to document what is common between the two major Smalltalk implementations. So, it leaves out what is not common. This points the way for future standards activities.

As additional implementations appear, they need to be compared to the common base. Decisions have to be made concerning what to do about existing incompatibilities. Many questions will be outside the scope of standardization, but some will need to be addressed. The impact and importance to users should be the determining factor.

The primary goal is to produce a language standard. The problem with doing this with Smalltalk is that it's not particularly clear where the language ends and class libraries take over. With the common base we made some conscious decisions:

1. We purposely avoided attempting to standardize user interface classes. The pragmatic reason is that this is where most of the differences lie between existing implementations. On the other hand, other language standards do not address user interface libraries. Smalltalk should not be penalized because it does not standardize areas not addressed by other language standards.
2. We purposely tackled higher-level language features, such as the collection classes, and some aspects of class objects, because these features make Smalltalk what it is.

Starting a standards effort inevitably triggers the desire to

continued on page 9...

PRODUCT ANNOUNCEMENTS

OBJECT THINK

Peter Coad and Jill Nicola have just completed a new book titled Object-Oriented Programming. The book teaches “object think,” the thinking stategies necessary for effective use of object technology. It also teaches how to program effectively using the two leading object-oriented programming languages: C++ and Smalltalk. The OOP book consists of four large examples: a counter, a vending machine, a sales transaction system, and a traffic flow management system. It introduces strategies and language details just at the moment each can be applied with success. According to Fotios Skouzos, IS Director at Falcon, “The OOP book has quickly become the most consulted desk reference within my development group.”

The book is available from Prentice Hall technical bookstores or directly from the authors at Object International.

Object International, Austin, TX
800.662.2557 or 512.795.0202 (v), 512.795.0332 (f)

OOP WORKBENCH FOR MACS

SmalltalkAgents for Macintosh is an object-oriented software development workbench and application delivery tool with advanced computing capability.

Based on a superset of the Smalltalk language, SmalltalkAgents has extensions patterned after C and LISP, and fully supports the Macintosh toolbox including traps and callbacks. It provides a powerful new set of tools which greatly increases a programmer's productivity. SmalltalkAgents possesses advanced computing capabilities such as dynamic linking, true preemptive interrupt-driven threads and events, transparent memory management, a 24-bit international character set supporting Unicode and Worldscript, and a rich class library. SmalltalkAgents requires a Macintosh with at least a 68020 CPU, 5 MB of RAM, and a hard disk. All features are fully functional with System 7 and 7.1, with limited support for System 6.0.7.

Quasar Knowledge Systems, Bethesda, MD
301.530.4858 (v), 301.530.5712 (f)

Highlights

Excerpts from industry publications

SPECIFICALLY SMALLTALK

One of the more significant happenings this year has been the emergence of Smalltalk as an application development environment for commercial application developers. American Airlines, for example, has deployed a commercial system to manage the resources required for all flights worldwide. This high-reliability, high availability distributed system was programmed in Smalltalk and is considered a major success.

1992 was also the year that Smalltalk companies got professional management . . . . The other challenge facing new professional managers of Smalltalk companies is that MIS directors can be very demanding to do business with. They demand services and insist upon delivering new products on or about the published schedules. As they evaluate Smalltalk, they see a lot missing. The challenge for the next couple of years will be to rapidly add capability without losing focus. Development environment companies should build strong development environments and kernel classes for their language . . . .

Just as Smalltalk has begun to creep into mainstream businesses, the harsh, cruel realities of using C++ as an application development language have been felt in company after company. while C++ can be used as an object-oriented language, it typically is not. Rather, it is used as a more complex C with esoteric new features that someday must be understood . . . .

A rather startling change has been in the paychecks of highly competent O-O designers and developers. Some have doubled; a few have tripled in the last year. Companies are beginning to recognize that someone who really knows existing object-oriented libraries and tools can be worth more than five greenhorns. For this time-to-market advantage, they are willing to pay handsomely. I have seen individual Smalltalk programmers working for $2,000 per day on long-term contracts and Objective-C programmers making a salary of $200,000 per year, and this trend will accelerate.”

MIS radar detects objects for the first time, Tom Love, HOTLINE ON OBJECT-ORIENTED TECHNOLOGY, February 1993

Current technologies for packaging class libraries have several problems; the most important is that they are highly lan-

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.

GRAPHICAL CLASS LIBRARY

ObjectBits 2.0 is a sophisticated class library that permits advanced programmers to create graphical applications effectively in the ObjectWorks\Smalltalk Release 4.1 environment. Programmers can understand it quickly and use it easily because it is implemented using purely Smalltalk technologies and methodologies. ObjectBits is implemented in a modular fashion and features components such as 2-D and 3-D charts, gauges, geometric figures, and bit and image editors. ObjectBits 2.0 is available on the Sun SPARCstation, HP 9000 series 70, IBM RS/6000, and Macintosh platforms.

Fuji Xerox Information Systems, Tokyo, Japan
81.3.3378.8284 (v), 81.3.3378.7259 (f)

GUI BUILDER FOR SMALLTALK APPS

Object Technology International (OTI) and Objectshare Systems have announced a new version of WindowBuilder that is integrated with ENVY/Developer. The two companies will also cooperate to ensure that future releases of their respective products are compatible.

The new version of WindowBuilder will be available in the format of an ENVY/Developer library. Previous versions of the two products required an integration effort by the customer before they could coexist in the same Smalltalk image. Customers will now be able to load and unload WindowBuilder into their ENVY/Developer environment with no additional effort.

ENVY/Developer is an integrated multiuser environment for large-scale Smalltalk development. It provides a highly productive team programming environment that supports the prototyping, development, release, and deployment of Smalltalk applications. The product's features include configuration management, version control, support for multiplatform development, performance profiling tools, a high-speed object storage and retrieval utility, and packaging tools for producing standalone executables.

WindowBuilder is the leading Smalltalk product for building graphical user interfaces. Developers can quickly construct sophisticated user interfaces for their end-user applications. The result is less manual programming and tedious layout when developing applications with windowing front-ends. WindowBuilder is available for Digitalk's Smalltalk/V for Windows and Smalltalk/V for OS/2.

Objectshare Systems, San Jose, CA
408.727.3742 (v), 408.727.6324 (f)

BUSINESS RE-ENGINEERING METHODOLOGY

CONSTRUCT is a leading-edge business re-engineering methodology that integrates all three facets of a business—strategy, operations, and information systems, to help companies manage change. CONSTRUCT is the first methodology to enable companies to define their fundamental purpose and ensure that all work performed in the organization has a demonstrable link to that purpose. In addition, CONSTRUCT is the only methodology that incorporates Business Works, an object-oriented software tousled developed by ParcPlace Systems that enables companies to refine strategy and rapidly translate it to every element of the business.

BusinessWorks is based on ParcPlace's VisualWorks, an ADE for creating graphical, client/server applications that are completely portable across PC, Macintosh, and UNIX systems. VisualWorks' database access capabilities allow developers to combine the power of hierarchical, relational, and object-oriented database systems with object-oriented programming technology for client/server applications. VisualWorks is based on ObjectWorks\Smalltalk.

Gemini Consulting, Morristown, NJ 07960
201.285.9000 (v), 201.285.9586 (f)

AUTOMATIC DOCUMENTATION TOOL

Synopsis for Smalltalk/V provides an automatic class documentation tool for development teams using Digitalk Smalltalk/V. The automatic documentation of Smalltalk classes allows development teams to eliminate the lag between the production of code and the availability of documentation. Using information already present in the Smalltalk/V environment, Synopsis automatically generates class documentation for any class in the system. Class documentation takes the form of a summary, made up of class comments, comments about variables, and documentation strings from class and instance methods. These summaries are similar to what you find in the Encyclopedia of Classes section of any Digitalk's Smalltalk/V manual.

With Synopsis, any effort by developers to improve class or method comments in the code is immediately reflected in the net class summary generated. Therefore, documentation lag time is minimized. In addition, documentation time is reduced because a large part of the work is done once during coding.

Synopsis Software, Raleigh, NC
919.847.2221 (v), 919.847.0650 (f)

CLASSIC SMALLTALK BUGS

Ralph Johnson

Every programming system is prone to certain kinds of bugs. A good programmer learns these bugs and how to avoid them. Smalltalk is no exception. Although Smalltalk eliminates many bugs that are common in other languages, such as bugs in linear search algorithms (just use do:), it has its own set of classic bugs, which most new Smalltalk programmers learn the hard way.

There are several reasons to collect classic bugs. First, it will help experienced programmers test and debug programs, and it can help us design better programs. Second, if we teach these bugs to novice Smalltalk programmers, they should learn to be good programmers faster. Third, perhaps we can redesign the system to eliminate some of these bugs, or we can write checking tools to spot them automatically.

I started the following list and posted it to comp.lang.smalltalk. Lots of people responded with more bugs, instructions on how to fix the bugs, and comments about my bugs. The result is the following list.

BUG 1: VARIABLE-SIZED CLASSES

Set, Dictionary, and OrderedCollection are variable-sized classes that grow. They grow by making a copy of themselves and "becoming" the copy. If you add new instance variables to a subclass, you have to make sure these instance variables get copied, too, or you will mysteriously lose the values of the instance variables at random points in time.

Smalltalk-80 R4.0 (and probably some earlier versions) has a #copyEmpty: method in Collection that you are supposed to override if you make a subclass of Collection that adds instance variables. The solution to this bug is to write a version of #copyEmpty: for your class.

It would be easy to write a tool that checked that every new subclass of Collection that added instance variables also defined a method for #copyEmpty:.

BUG 2: #ADD: RETURNS ITS ARGUMENT

Most collections that grow implement the #add: method, which returns its argument. Most new Smalltalk programmers assume that #add: returns its receiver, which leads to problems. Thus, they write "(c add: x) add: y" when they should really write "c add: x; add: y" or "c add: x. c add: y". This is one of the good uses for #yourself. For example, you can write:

```
(Set new
    add: x;
    add: y;
    ...;
    yourself)
```

to make sure that you have the new Set.

#add: returns its arguments for serveral good reasons. Making #add: return its argument often keeps you from resorting to temporary variables, because you can create the argument to #add: on the fly and use the argument afterward. If you want to access the collection, you can do it with #yourself and cascaded messages, as described above.

Nevertheless, after years of explaining how #add: works to students, I wish that it had been defined to return its receiver. It is too late to change now without confusing every Smalltalk programmer on the planet, so it is a problem we have to live with.

BUG 3: CHANGING COLLECTION WHILE ITERATING OVER IT

Never, never, never iterate over a collection the iteration loop modifies. Elements of the collection will be moved during the iteration, and elements might be missed or handled twice. Instead, make a copy of the collection you are iterating over. That is, aCollection copy do: [:each | aCollection remove: each] is a good program, but if you leave out the copy it isn't.

Mario Wolczko suggested a solution that catches this problem the instant it occurs (at some performance penalty, of course). The solution is to change the collection classes. Each iteration method enters that collection into a set of collections being iterated over (IteratedCollections), executes the block, and then removes the collection from the set. Collections are usually modified using #at:put: or #basicAt:put:, so these are overridden to check that the collection is not in IteratedCollections. If it is, an error is signaled. You can either use this technique all the time or just install these classes when you are testing and debugging your program. The changes are packaged in a file called Iterator-check.st that is available on the Manchester and Illinois servers. On the Illinois server, it is in pub/MANCHESTER/manchester/4.0/Iterator-check.st.

BUG 4: MODIFYING COPIES OF COLLECTIONS

It is common for an object to have an accessing method that returns a collection of objects you can modify. However, sometimes an object will return a copy of this collection to keep you from modifying it. Instead, you are probably supposed to use messages that will change the collection for you. The problem is that this is often poorly documented, and anyone who likes to modify collections directly will run into problems. See "ScheduledControllers scheduledControllers" for an example.

The solution is to provide better documentation, to claim that nobody is allowed to modify copies of collections returned from other objects, or to have objects that don't want their collections modified to return immutable versions of the collections that will give an error if you try to modify them.

BUG 5: MISSING ^

It is very easy to leave off a return caret on an expression. If there is no return at the end of a method, Smalltalk returns the receiver of the method. It only takes one missing return to mess up a long chain of method invocations.

BUG 6: CLASS INSTANCE CREATION METHODS

Writing a correct instance creation method is apparently non-trivial. The correct way to do it is to have something like:

```
new
    ^super new init
```

where each class redefines #init to initialize its instance variables. In turn, #init is defined as an instance method init:

```
super init "to initialize inherited instance variables"
"initialize variables that I define"
```

> “It only takes one missing return to mess up a long chain of method invocations.”

There are lots of ways to do this wrong. Perhaps the most common is to forget the return, that is, to write:

```
super new init
```

As a result, you have the class where you want the instance of the class. This is a special case of bug number 5.

Another error is to make an infinite loop by writing:

```
^self new init
```

If Smalltalk doesn't respond when you think it should, press ^C to get the debugger. If the debugger shows a stack of #new messages, you know you made this mistake.

Finally, you should define #new only once for each class hierarchy and let subclasses inherit the method. If you redefine it in each class, you will reinitialize the new object many times, wasting time and perhaps memory.

One way to keep this from happening is to make the #new method in Object send #init, and have the #init method in Object do nothing. Of course sometimes the version of #init that you define has arguments, and this wouldn't help those cases. It is probably better to rely on education to eliminate this kind of error.

BUG 7: ASSIGNING TO CLASSES

OrderedCollection := 2 is perfectly legal Smalltalk, but does dreadful things to your image.

This bug could be eliminated if the compiler gave a warning when you assign to a global variable that contained a class.

BUG 8: BECOME:

#become: is a very powerful operation. It is easy to destroy your image with it. Its main use is in growing collections (see bug number 1), since it can make every reference to the old version of a collection become a reference to the new, larger version. It has slightly different semantics in Smalltalk/V and Smalltalk-80, since x becomes: y causes every reference to x and y to be interchanged in Smalltalk-80, but does not change any of the references to y in Smalltalk/V.

Suppose you want to eliminate all references to an object x. Saying x becomes: nil works fine in Smalltalk/V, but will cause every reference to nil to become a reference to x in Smalltalk-80. This is a sure calamity. You want x to become a new object with no references, such as in x becomes: String new.

BUG 9: RECOMPILING BUGS IN SMALLTALK/V

It is easy to have references to obsolete objects in Smalltalk/V if you change code without cleaning things up carefully. For example, the associations whose keys are the referenced names in the Pool Dictionary are stored in the CompiledMethods at compile time. If you create a new version of the Pool Dictionary and install it by simple assignment, the compiled methods still refer to the old associations.

If you substitute a new instance of Dictionary or replace, rather than update an association in a pool dictionary, you have to recompile all methods using variables scoped to that Pool.

This is is also annoying when using ENVY, where the methods are under strict control. Perhaps Pool Dictionaries should be be first-class versioned prerequisites of classes, just like the class definition.

If you prune and graft a subtree of your class structure, you have to make sure that all referencing methods are recompiled. Otherwise, you (or your customer, because this is only detected at runtime) will run into a Deleted class error message. Thomas Muhr posted a "bite" a while ago to handle this problem for Smalltalk/V 286.

BUG 10: OPENING WINDOWS

Older versions of Smalltalk/V and Smalltalk-80 do not return to the sender when a new window is opened. Thus, any code after a message to open a window will never be executed. This is the cause of much frustration. For example, if you try to open two windows at once, that is:

```
TextPane new open.
TextPane new open
```

in Smalltalk/V 286 and

```
aScheduledWindow1 open.
aScheduledWindow2 open
```


YOU CAN DO IT WITH EXCEPTIONS

An exception handling mechanism is built to handle just these sort of cases, breaking out of normal processing to handle some special condition. ParcPlace Smalltalk has one integrated with the language, and there are several implementations available for Digitalk versions.

Hubert Baumeister (hubert@mpi-sb.mpg.de) provides a detailed example of how to do this. We can define a signal handler as:

```
LoopBreakSignal := Signal genericSignal
        notifierString: 'Using break without being in a loop';
        nameClass: self message: #loopBreakSginal.
```

repeat a block using:

```
Context>>loop
        "Evaluate the receiver repeatedly, not ending unless 'Object
        loopBreakSignal' is raised or the block forces some stopping
        condition, like method returns, Signals raised but not handled
  etc.."
        Object loopBreakSignal handle: [:exp | ]
                do: [self repeat]
```

and then invoke it with the Object method

```
break
        LoopBreakSignal raise.
```

This is very similar to the use of exceptions for handling assertions, which was discussed in this column in the October 1992 issue. This is nicer, since we don't have to change any system classes, but it still has a couple of disadvantages.

First, it makes the code for looping a bit more complicated, and if we want it to be available everywhere we have to modify system methods like do:. If we want the block to return a value, we have to do even more complicated things. It probably has a fairly substantial overhead. Finally, and most important, it could lead to very confusing results.

Exception handling is a very general facility for handling non-local control transfers. It can be used to implement a facility for breaking out of a loop, but in complicated cases, the programmer needs to have the discipline to ensure that control is being transferred to the intended place.

YOU CAN DO IT WITH BLOCKS

A cleaner solution also uses the method returning mechanism, but to pass a method return as part of another block.

Ralph Johnson (johnson@cs.uiuc.edu) describes this as follows:

> **There are lots of ways to break out of a loop. The important thing to realize is that the only ways to change control flow in Smalltalk are to send a message and to return from a message, but blocks let you treat code as data and so control where you are going to send a message.**
>
> **The result of the above is that to simulate a go-to, you have to introduce extra blocks. For example, here is a simple way to break out of a loop:**

```
[obj foo]
        whileTrueWithBreak:
        [:exit |
                "loop body is here"
                timeToLeave ifTrue: [exit value].
                "finish up loop"]
```

whileTrueWithBreak: is defined in BlockClosure (in 2.5-4.1, BlockContext in 2.3, and I don't know where in Smalltalk/V) to be:

```
        whileTrueWithBreak: aBlock
                ^aBlock value: [^nil]
```

> “Smalltalk blocks are most often used as simple control structures, and we usually don't have to think about their full capabilities.”

Mario Wolczko also advises that the Manchester goodies library has similar code in the BlockWithExit goodie. The library is accessible by ftp@st.cs.uiuc.edu or at mushroom.cs.man.ac.uk.

This kind of code can be very confusing. Smalltalk blocks are most often used as simple control structures, and we usually don't have to think about their full capabilities. In this case, we pass as an argument a block that returns *from the method context in which it was defined.* Although there may be a great deal in the stack below that point, it is immediately discarded, and we resume execution at the next level up from that defining method.

This is quite a neat trick. It breaks out of a loop without using any additional language mechanisms, and it makes the code only a little uglier.

Unfortunately, to handle return values nicely, we have to add a bit more ugliness, adding a parameter to the exit block.

```
whileTrueWithBreakReturningAValue: aBlock
    ^aBlock value: [:returnValue | ^returnValue].
```

Writing a more complicated loop, like injectWithBreak:into: can start to get complicated. For one thing, the block will require three arguments, which is a problem in Digitalk dialects. Also, like exceptions, blocks can provide much more general transfers of control, and the programmer must ensure that the results are correct.

WHAT'S THE BEST WAY?

Considering that you can't return from a block in Smalltalk, there are a lot of different ways of doing it. Unfortunately, they all have their drawbacks. **Ralph Johnson** comments:

continued on page 23...

THE BEST OF comp.lang.smalltalk

Alan Knight

Breaking out of a loop

This month's discussion started with a question from Deeptendu Majumder (gt0963d@prism.gatech.edu), who writes:

> I [have] always found a way to avoid this, but I would like to know how to break away from inside a loop and return [to] the immediate-upper-level context.

Although this question may seem elementary to an experienced Smalltalker, and the straightforward answer is probably the best, I found the wide variety of answers worthwhile and a reminder of how many different ways things can be accomplished in Smalltalk.

Unfortunately, the first answer that comes to mind is to dismiss the question.

FORGET IT

The language doesn't provide it, but it's easy to work around. Anyone who didn't just fall off the cabbage truck knows that. Next message.

This is an effective attitude for getting through news quickly, but it's not very helpful. The least we can do is describe the standard workaround.

HERE'S WHAT YOU DO INSTEAD

The obvious answer is that, although you can't break out of a block prematurely, you can break out of a method. By pushing the loop into a separate method, you can use the normal return mechanism.

For example, suppose we have a method like:

```
SomeClass>>someMethod
  self startUp.
  collection do: [:each |
        each doSomething.
        self specialExitCondition
                ifTrue: ["Break, but still do the finish up code"].
        each doSomeMore].
  self finishUp.
```

We can't break out of the loop and still do the finishUp code. To make it work, we need to break it into two methods.

```
SomeClass>>doSomething
  self startUp.
  self loop.
  self finishUp.

SomeClass>>loop
  collection do: [:each |
        each doSomething.
        self specialExitCondition ifTrue: [^self].
        each doSomeMore].
```

When specialExitCondition is true, we return from the loop method, but still execute the finishUp code. It's a simple transformation on code, and breaking the code into smaller pieces this way often improves it. Who could ask for more?

Well, perhaps it improves the code, but I doubt that it always does. While decomposing code into smaller pieces is usually good, I'd much rather do it along logical lines than along lines imposed by the language.

YOU CAN DO IT IF YOU'RE CLEVER

Saying that Smalltalk can't do something is often a mistake, particularly when you are in a virtual room with a lot of clever programmers.

Jan Steinman (steinman@hasler.ascom.ch), who is well-acquainted with the inner workings of Smalltalk, writes:

> It is possible, but it is ugly. I had implemented it in Tek Smalltalk for "real" blocks, via a Context stack hack, but I haven't tried to make it work with 4.1 BlockClosures. It would necessarily change the semantics of blocks somewhat—what does the block answer when "broken," for instance?
>
> Then there's the case of in-line "pseudo-blocks." My context stack hack never did work with compiled in-line blocks, like #to:do:. This is a real problem, since the system goes out of its way to hide the difference from you!
>
> To make it work with pseudo-blocks might actually be easier. It would take a compiler hack that would simply jump out of the loop. But then the semantics would be different than for breaking out of a real block via a stack unwind mechanism. Yuk.

So it can probably be done if we're sufficiently clever. This is fascinating for dedicated Smalltalk hackers and for language designers, but I don't think it's a good answer for a novice or for somebody who just wants to get things done. It would be easier to just rework the code as in the previous section. Is there a better way?

in Smalltalk-80, then you will get one open window and one forgotten piece of code. This problem has been fixed in Objectworks\Smalltalk R 4.1 and later releases of Smalltalk/V, so the above code will create two windows as you would expect.

The fix for earlier versions of Smalltalk-80 is to use the openNoTerminate method to open the window, which does not transfer control to it. A useful trick is to store the new window in a global variable so you can test it.

Aad Nales says that the fix for Smalltalk/V286 is to fork the creation of the new window:

```
[Textpane open] fork.
```

If this is not what the programmer wants, it is probably necessary to hack the dispatcher code and remove the dropSenderChain message, which is the ultimate cause of the problem.

BUG 11: BLOCKS

Blocks are powerful, and it isn't hard for programmers to get into trouble trying to be too tricky. To compound problems, the two versions of Smalltalk have slightly different semantics for blocks, and one of them often leads to problems.

Originally blocks did not have truly local variables. The block parameters were really local variables in the enclosing method. Thus:

```
| x y |
    x := 0.
    (1 to: 100) do: [:z | x := x + z]
```

actually had three temporaries, x, y, and z. This leads to bugs such as the following:

```
someMethod

    | a b |
    a := #(4 3 2 1).
    b := SortedCollection sortBlock: [:a :b | a someOperation: b].
    b addAll: a.
    Transcript show: a.
```

When elements are added to b, the sortBlock is used to tell where to put them. What gets displayed on the transcript will be an integer, not an array.

Early versions of Smalltalk-80 (2.4 and before) implemented blocks like this, and Smalltalk/V still does. However, in current ParcPlace implementations, blocks are close to being closures. You can declare variables local to a block, and the names of the block parameters are local to the block. Most people agree that this is a much better definition of blocks than the original one. Nevertheless, people planning to use Smalltalk/V should realize that it has a different semantics for blocks.

This difference can lead to some amusing problems. For example, here is some code written by someone who had obviously learned Scheme:

```
| anotherArray aBlockArray |

aBlockArray := Array new: 4.
anotherArray := #(1 2 4 8).
1 to: 4 do: [ :anIndex |
    aBlockArray at: anIndex put: [ (anotherArray at: anIndex) * 2 ]].
```

The programmer expected each block to be stored in the array along with its own value of anIndex. If anIndex were just a local variable of the method, this will not work. It assumes that each execution of the block gets its own version of anIndex, and Smalltalk/V and old Smalltalk-80 actually make each execution share the same version.

So, if you are using Smalltalk/V, be careful not to reuse the names of arguments of blocks unless you know that the blocks are not going to have their lives overlap. Thus:

```
aCollect do: [:i | ...].
bCollect do: [:i | ...].
```

is probably OK because #do: does not store its argument, so the blocks will be garbage by the time the method is finished. However, if the first block were stored in a variable somewhere and evaluated during the execution of the second block then problems would probably occur.

BUG 12: CACHED MENUS

Menus are often defined in a class method, where they are created and stored in a class variable or a class instance variable. The method will look something like this:

```
initializeMenu
    ...
```

Note that accepting the method does *not* change the menu. You have to execute the method to change the class variable or class instance variable. Often the #initializeMenu method is invoked by the class method #initialize. This can lead to the strange effect that you can initialize the menu by deleting the class and filing it in again, but otherwise you don't seem to be able to change the menu (because you haven't figured out that you should really be executing the #initializeMenu method).

To make matters worse, it is possible that each instance of the controller, or model, or whatever has the menu, stores its own copy of the menu in an instance variable. If that is the case, it is not enough to execute #initializeMenu, you must also cause each object to reinitialize its own copy of the menu. It is often easier to delete the objects and recreate them.

Often a class will have a #flushMenus method to clear out all menus. Typically the method that fetches the menu will check to see if it is nil and invoke #initializeMenu if it is. So, #flushMenus will just "nil out" the variable holding the menu. The best way to figure out what is happening is to look at all uses of the variable. Smalltalk experts rarely have problems with this bug, but it often confuses novices.

Caching is a very common technique in Smalltalk for making programs more efficient in both time and space. Caching of menus is one of the simplest uses of caches, and other uses can create more subtle bugs.


```
Object>>isSpecialized

    ^self methodDictionariesField == self class methodDictionaries
```

Next come the methods for actually specializing the receiver. The first sets up an array with a fresh MethodDictionary.

```
Object>>specialize
    | old new |
    self isSpecialized ifTrue: [^self].
    old := self methodDictionariesField.
    new := (Array with: (MethodDictionary newSize: 2)) , old.
    self methodDictionariesField: new
```

The next one takes a string, compiles it, and installs the result in the private dictionary:

```
Object>>specialize: aString
    | association |
    self specialize.
    association := Compiler compile: aString in: self class.
    self methodDictionariesField first add: association
```

CONTRASTS

What do these two implementations of instance specialization say about their respective systems? For one thing, both of them are simple, clean, and easy to understand. The external protocol is exactly the same. There isn't much to choose from between them. From that standpoint, I would have to say that both systems support a fairly esoteric change to the language semantics with a minimum of fuss.

The ParcPlace implementation is conceptually cleaner to me. The user's model that the behavior of an object is always defined by its class is retained. It's just a little easier to create classes than you thought. The Digitalk implementation requires that you understand the particular mechanism they have lying behind that conceptual model so that you can implement the necessary changes.

When I understood the ParcPlace implementation I said, "Ah, that makes sense." When I understood the Digitalk implementation I said, "Cool! That really works?" The ParcPlace model is an extension of the semantics. The Digitalk model is an extension of the implementation.

I am fishing for just the right way to characterize the difference. I don't think I can make it clear yet, but I also don't think it will be the work of a single week, or even a single year, to make it clear. Let's barrel on.

As you get to know both product lines, you will find this same distinction repeated many times. I think that the difference stems from the diverging goals of the technical luminaries at the two companies. The ParcPlace image was driven first by Dan Ingalls and then by Peter Deutsch. Both have strongly developed aesthetic sensibilities to go along with their amazing technical skills. A solution wasn't a solution to them until it was beautiful. Actually, now that both of them have gone on to other things, the ParcPlace models are beginning to show signs of creeping cruft.

Jim Anderson and George Bosworth, on the other hand, are primarily motivated by the belief that software just shouldn't be that hard to write. They produced Smalltalk/V so others could write software more easily. Their success criteria seems to be "if it's better than C, it's good enough." They weren't about to let a little thing like a less-than-perfect conceptual model get in the way of shipping product. Of course, they had a company to run as they were developing their image, unlike ParcPlace in the early (Xerox PARC) years, so they didn't have much choice about the importance of aesthetics.

> “Hmmm… different implementations, same interface—maybe this object stuff works, after all!”

Don't take this to mean that the ParcPlace image is truth and beauty personified and the Digitalk image is a baling-wire-and-chewing-gum collection of dire hacks. There are areas where each beats the other in both conceptual model and implementation. However, I think it is safe to say that the primary motivations behind the two systems are a contrast between aesthetics and pragmatism.

What this means for the workaday programmer isn't entirely clear. Most of the time, the ParcPlace image provides smooth development. Every once in a while, though, you will encounter a good idea that hasn't been taken quite far enough, and you will have to bend yourself into a pretzel or bypass it entirely to get around it. Put another way, if you are going the ParcPlace way you will have lots of support. If, however, you have the misfortune to want to do something a different way than the original implementor imagined, you may be in trouble. In these cases you will often have to browse around and understand lots of mechanism before you can figure out how to wedge your code in.

The Digitalk world is less coercive, but it's also less supportive. For code that relies heavily on their implementations (i.e., not just instantiating collections) I average more lines of code to get the same functionality. I know there have been cases where the Digitalk implementation has been easier. I don't think a Digitalk project has ever been conceptually simpler, though.

In future columns, I will explore more specifics of the contrast between the systems, and try to quantify why one or the other is better for specific tasks. In the meantime, if you run into situations that are surprisingly hard or easy in either sys-

continued on page 23…

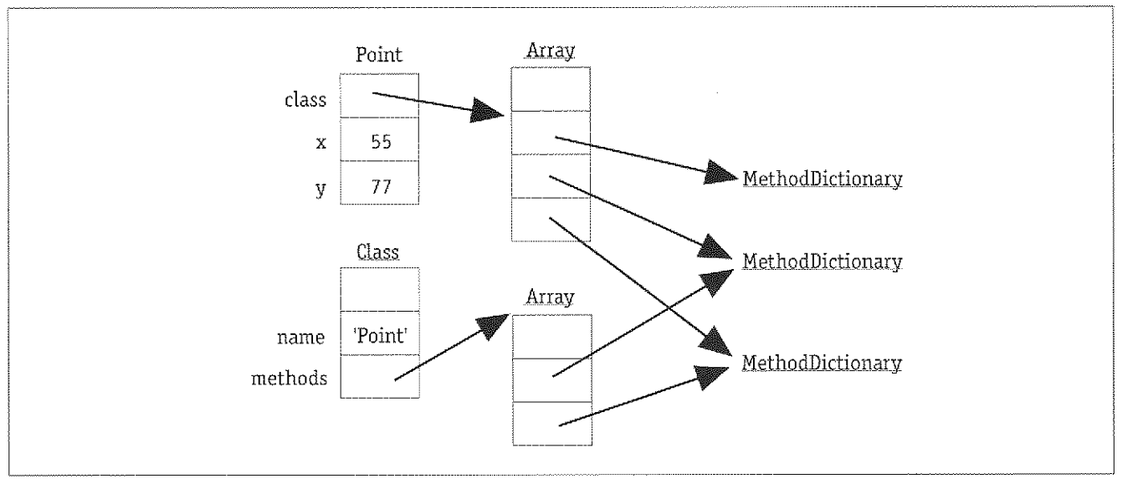


Figure 2. A specialized Point.

Conceptual Model

What's so special about the class constructing the array? It's just an Array whose elements are MethodDictionaries. Any object can build one of those. That's how we'll implement instance specialization. We'll fetch the array that's there and copy it, adding a slot at the beginning containing a fresh MethodDictionary. Then we can make all the changes we want to the private MethodDictionary without affecting any other instances.

Example

Before we can implement the conceptual model we need access to a couple of hidden primitives to get and set the method dictionaries field of the object.

```
Object>>methodDictionaryField
    "Return the Array of MethodDictionaries for the receiver"
    <primitive: 96>
    self primitiveFailed

Object>>methodDictionaryField: anArray
    "Set the Array of MethodDictionaries for the receiver
    to anArray. anArray must contain MethodDictionaries
    or your system will crash!"
    <primitive: 97>
    self primitiveFailed
```

Now we need to get something on the screen to see the effects of our experiments. Fortunately, that's easy in Smalltalk/V.

```
TopPane new open inspect
```

When we execute the above expression we get a window and an inspector on that window. In the inspector we can execute the following to get a fresh MethodDictionary to put our specialized methods in:

```
| old new |
old := self methodDictionaryField.
new := (Array with: (MethodDictionary newSize: 2)) , old
self methodDictionaryField: new
```

Now we can specialize our window by executing the following in the inspector:

```
| association |
association := Compiler
    compile: 'display Transcript show: "Howdy". super display'
    in: self class
self methodDictionaryField first add: association
```

Now if you execute self display you will see that, indeed, the specialized method is being invoked. (You will have to send the window backColor: for the superclass' display method to work).

Methods

I was surprised at how easy it was to implement instance specialization methods that were compatible with the ParcPlace version. I had expected the differences in implementation to leak through into the interface. Hmmm . . . different implementations, same interface—maybe this object stuff works, after all!

The first method I defined last time was one you would duplicate in any class in which you wanted all instances to be specializable. I don't think this is necessary, since the lazy specialization implemented below works fine. For completeness, though, here it is:

```
new
    ^super new specialize
```

The method I defined in the last issue should have been defined this way, rather than duplicating the specialization code in the class and the instance. I think I did it the way I did because that was how I saw it first implemented by Ward Cunningham when he put scripts into HotDraw.

Next is a method to test whether an instance is ready to be specialized. Since all unspecialized instances of a class share the same array of dictionaries, if the receiver has a different array we will assume it has a private array.

BUG 13: SINGLETON OBJECTS IMPLEMENTED WITH CLASS METHODS

Sometimes you need to make a globally known object that is the only member of its class. These singleton objects are sometimes implemented as class methods and class variables. This works fine in the short term, but does not work in the long term because the time inevitably comes when you need to make more instances of the class. If you have implemented an object with class methods, you will have to rewrite the class or try to implement a second object by making a subclass of the first.

> “Blocks are powerful, and it isn't hard for programmers to get into trouble trying to be too tricky.”

The correct way to implement a globally known singleton object is to make a normal class for it, to define a class instance variable to hold the singleton object (in Smalltalk-80 this is done in the definition pane of the browser when the “meta” button is pressed), and to have a class method (I like the name #default) return the value of the variable, initializing it if it is nil. This is like a cache, and nearly eliminates the possibility of an initialization error.

Another alternative is to make a singleton object be the value of a global variable. There is no other proper use of global variables. Storing an object in a global variable is proper when there are instances of the class used for other purposes. For example, the global variable Undeclared in Smalltalk-80 is just a regular Dictionary. However, it is probably not a good way to implement a singleton class, because making sure that a global variable is initialized is a common source of problems.

CONCLUSION

I would like to thank the many people who contributed bugs or solutions to bugs to the list: Amir Bakhtiar, Hubert Baumeister, Naci Dai, Marten Feldtmann, Peter Goodall, Alan Knight, Simon Lewis, Eliot Miranda, Thomas Muhr, Aad Nales, Kurt Piersol, Jan Steinman, Mario Wolczko, Mike Smith, Terry Raymond, Dave Robbins, Randy Stafford, Michael Sullivan, Brent Sterner, Nicole Tedesco, Rik Fischer Smoody, and Markus Stumptner.

If you would like to bring bugs to my attention, please post them to comp.lang.smalltalk, email them to me at johnson@cs.uiuc.edu, or write me at Department of Computer Science, 1304 W. Springfield Ave, Urbana, IL 16801. ▣

Ralph Johnson is affiliated with the University of Illinois at Urbana-Champaign.

...continued from page 4

“improve” the language. Although this desire is good, we think that the overriding goal must be to achieve a common specification that is supported by available implementations. While this is likely to require some compromise between the various Smalltalk implementors and the constituents of the user community, we believe the ultimate arbiter should be the Smalltalk user community. The users are the ultimate audience for Smalltalk and the standard.

CONCLUSION

Smalltalk is more than 10 years old. It has come a long way in overcoming the perception of being a research language and has entered the realm of commercial application development. We believe a standard is needed, and the time is now. If you agree, please encourage your organization to join us in ANSI to define the standard. Together, as Smalltalk users, we can ensure our success and contribute to the acceptance of Smalltalk by the software development community at large. ▣

Acknowledgments

We would like to thank Digitalk, KSC, OTI, and ParcPlace for their contributions to and support for the project. We would also like to thank all the IBM internal reviewers, the legal and contract team, ITSC and its editors, and our management for supporting this effort.

Reference

1. Cook, W. Interfaces and specifications for the Smalltalk-80 collection classes, PROCEEDINGS OF OOPSLA '92, pp.1–15.

Rick DeNatale is a Senior Programmer with the IBM Systems Laboratory in Cary, NC. In 1993, he headed a team that designed and implemented a hybrid O-O language called ClassC. He is a co-author of the Smalltalk Common Base document. He can be reached by email at denatale@carvm3.vnet.ibm.com.
Y.P. Shan is a Development Staff Member at the IBM Systems Laboratory in Cary, NC. He has been active in researching and developing object-oriented technology since 1986. He can be reached by phone at 919.469.6571, fax at 919.469.6948, or email at shan@carvm3.vnet.ibm.com.

PUTTING IT IN PERSPECTIVE

Rebecca Wirfs-Brock

The incremental nature of design

It is good to have an end to journey towards;
but it is the journey that matters in the end.
—Ursula K. LeGuin

Design requires effort, review, reflection, and rework. I don't know of anyone who has built an application right the first time. Objects always need rework and redefinition. Solutions should remain fluid throughout an incremental design and implementation. In this column, I want to reflect on when a design starts and when it is finished. I also want to touch on some major differences between incremental design and implementation cycles and rapid prototyping.

HOW DESIGN REALLY WORKS

Designing object software means creating an executable model of interacting objects. One fundamental difference between software design and software analysis is that designs have to be translated into working programs. Analysis results need to reflect an accurate statement of the problem and constrain possible solutions, but they don't have to work. We designers still have to solve the problem. Solving even a well-defined problem is not always straightforward or easy.

I find software design to be inherently messy and fraught with mistakes. It involves top-down, bottom-up and sideways building and rebuilding of a solution. I try to teach this to my design students while giving them a strong foundation for building object designs. Designers and implementers appreciate this honest exposure to the way things really work and are eager to pick up some immediately useful skills they can apply to object design.

I've had managers sit in on design sessions (or even worse, in classrooms) and get very concerned that designers aren't honing in quickly enough on the "right" solution. Besides hindering progress, this can be demoralizing to teams new to object design. I've also worked with managers who entrust teams from the start to solve problems and produce results. Only when a schedule appeared to be in jeopardy or the team called for help did they get concerned. The enthusiasm and positive energy that sparks a team having this style of leadership are amazing! The key to these managers' success, in my opinion, was that they empowered design teams while imposing plenty of non-threatening process checks along the way.

The American Heritage Dictionary defines design as "plan[ning] out in a systematic . . . form." I like this definition. It characterizes design as systematic planning. We're still error-prone, even if we are systematic about software design. Is that the fault of the designers, their tools, or the imprecision of inputs to the process? I don't think we should place blame on any of these factors. We software designers are inherently building complex systems. Although some researchers are actively investigating better ways to precisely state requirements while others are working at ways to minimize the transformations we make between software analysis and software design, we designers and implementers still have to deal with unpredictability. Unless we are rebuilding a system for the *n*th time, we will continue to discover additional constraints throughout implementation.

Object technology improves our chances of building well-designed systems. We have conceptual tools that help us decompose the problem. We can find objects in the problem domain that have representations in our executable programs. We can encapsulate functionality and data into objects to build high-level abstractions. Well-designed objects enable us to deal with increasing levels of complexity. Even so, we still haven't changed the bumpy, uneven nature of software development.

INJECTING DESIGN INTO IMPLEMENTATION

While software development isn't a smooth process, we still need a design process. Building systems more predictably demands that we interleave design throughout implementation. We need to consciously expend some fraction of our energy designing and refining our solution. Design needs to naturally occur throughout development. The alternative is to simply fix things so they work, or hack more functionality without considering the impact on future developers or system flexibility.

Incremental design means progressing toward a working solution in a planned fashion. One way to make orderly progress is to decompose design and implementation into a series of many small, inherently more manageable steps. I don't view incremental design as a heavily regulated or tightly monitored activity. I don't want to restrict forward progress or put a crimp on individual creativity. Designing involves an element of understanding how things work now while not accepting the status quo. Responsible designers take a broad perspective. It isn't enough to build the software; you also need to pay attention to the flexibility and elegance of the emerging solution.

Design doesn't come together at the end of a long design cycle and remain sacrosanct throughout implementation. In incremental development, systems aren't designed or integrated according to the Big Bang Theory. There are many small

SMALLTALK IDIOMS

Kent Beck

Instance specific behavior:
Digitalk implementation and the deeper meaning of it all

In the last issue, I wrote about what instance-specific behavior is, why you would choose to use it, and how you implement it in Smalltalk-80 . . . er . . . Objectworks\Smalltalk (which way does the slash go, anyhow?) . . . er . . . VisualWorks (is that a capital *W* or not?). This month's column offers the promised Digitalk Smalltalk/V OS/2 2.0 implementation (thanks to Mike Anderson for the behind-the-scenes info) and a brief discussion of what the implementations reveal about the two engineering organizations.

I say "brief discussion" because as I got to digging around I found many columns' worth of material there for the plucking. I'll cover only issues raised by the implementation of classes and method look-up. Future columns will contrast the styles as they apply to operating system access, user interface frameworks, and other topics.

DIGITALK IMPLEMENTATION

Runtime Structures

The Digitalk implementation of method look-up is slightly different from the ParcPlace model. Actually, until Smalltalk/V OS/2 2.0 (hereafter VOS2) the models were quite similar. The Digitalk implementation did not allow you to create Behaviors and instantiate them easily, so the instance specialization implementation presented in the last issue wouldn't work, but the pictures of the objects would have been identical.

The VOS2 model departs from the "classic" by giving each instance a reference, not to its class, but to an Array of MethodDictionaries (see Figure 1). In the normal case, the class constructs this array and all instances share it.

The ParcPlace implementation requires an additional indirection to reach the method dictionary, as the virtual machine has to go from the object to the class, and from the class to the method dictionary. With the VOS2 model, the virtual machine just has to go from the object to the array. Going up the superclass hierarchy is also faster, as the virtual machine can just march along the array rather than trace references from class to superclass.

Performance is not the primary motivation behind this design, however. More important, given the lack of flexibility in the implementation of Behavior and Class, this design makes it possible to specify the behavior of objects in many ways. For example, implementing multiple inheritance (ignoring different instance layouts in different classes) is simple. The class is welcome to create the array of method dictionaries any way it wants.

You may be wondering how the message "class" is implemented given the objects above. Each MethodDictionary has an additional instance variable called class, which is set to the class where it belongs (each class "owns" one and only one dictionary). The primitive for class marches along the array of dictionaries until it finds one whose class instance variable is non-nil, and returns that. That way, you can have dictionaries that don't belong to any class, and the scheme still works.

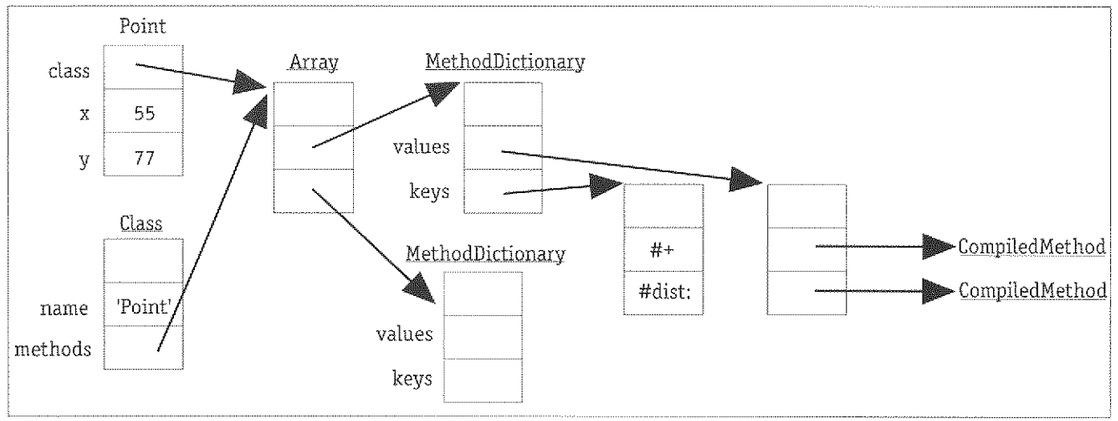


Figure 1. VOS2 objects supporting method lookup.

represents detailed status information about a file. An alternate solution is to create a class, called FileInformation to store this data. FileInformation has a class method to create new instances, and instance methods to access its components. A partial class specification follows:

file Information methods

fromfileEntry: afileEntry
Create and return an instance of the receiver for a file entry

file Information methods

fileName
Return the name of the file.

size
Return the size of the file, including both the data and resource fork.

timeStamp
Return the date and time when the receiver was last modified.

resourceSize
Return the size of the resource part of the file.

creatorType
Return the code that indicates the application that created the file.

With the FileInformation class, we can eliminate the use of Array and incorporate usage of our new class. The formatted method now looks like:

```
Directory methods
    formatted
        "Answer a collection of file information, one for each entry in
        the receiver."

            | answer file Entries anArray |
            file Entries := self contents.
            answer := Ordered Collection new: file Entries size.
            file Entries do:[ :each |
                    answer add: (FileInformation fromFileEntry: each)].
                    ^ answer
```

Clients of this method can then use meaningful selectors instead of indexing into an array. This code is more maintainable now and doesn't need any extra commenting.

```
| zeros |
zeros := myDirectory formatted select: [:info | info size = 0].
^zeros collect: [:info | info fileName]
```

There are good examples of Array use in your Smalltalk system. These are uses in which the index is a relevant part of the data structure, such as a numeric id allocated by the operating system. The array contains the relationship between the id and a related Smalltalk object. Literal arrays are convenient for collections of values.

IDENTIFYING INAPPROPRIATE USE

You can look for inappropriate use of Array and other data structures in your image. Use these techniques to find methods that reference Array. You may also want to look for references to other data structures such as OrderedCollection.

- In Team/V: Select Array in the Package Browser. Select the menu item Class/BrowseRefs.
- In Smalltalk/V for OS/2 and Smalltalk/V Windows: Execute Smalltalk senders Of: (Smalltalk associationAt:#Array)
- In Smalltalk/V Mac: Execute Smalltalk referencesTo:#Array.
- In Objectworks\Smalltalk: Select Array in the System Browser. Select the menu item Class Refs from the class pane menu.

> “Don't use arrays as a shortcut to pass around related items. Instead, create a class to represent the abstraction relating the items.”

When examining a method, inappropriate use will have one or more of the following characteristics:

- Indices that are irrelevant to data and functionality.
- Array elements that are related by some abstraction *not* captured by a class.
- Awkward client use due to violation of information hiding and encapsulation.

If you find a method that uses arrays inappropriately, you should improve the quality of your code by:

1. Creating classes to represent related array elements.
2. Rewriting offending methods to reference new classes and to eliminate arrays.

CONCLUSION

Don't use arrays as a shortcut to pass around related items. Instead, create a class to represent the abstraction relating the items. Your code will immediately be more understandable, extensible, maintainable, and reusable. Classes are the basic building blocks of Smalltalk programs. Use them. ▣

Juanita Ewing is a senior staff member of Digitalk Professional Services. 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 class libraries for the first commercial-quality Smalltalk-80 system.

cycles of discovery, design of a partial solution, analysis of the results, and rebuilding a better solution.

What distinguishes incremental design from rapid prototyping is this analytical step. *Analyze* means to "separate into parts or basic principles so as to determine the nature of the whole, to examine methodically." This is crucial to incremental design. Progress needs to be measured, reflected upon, and reviewed with others periodically. There is an openness on the part of the designer to change and improve.

Another characteristic that distinguishes incremental design from rapid prototyping is the willingness on the part of an incremental designer to throw out a bad design, rethink the problem, and redesign a solution.

The primary goal during rapid prototyping is to simply get it working. Many times an implementer during rapid prototyping knowingly (and quite possibly with some discomfort) builds something that is definitely not cleanly structured. It takes a lot of discipline to stop and clean things up with rapid prototyping.

Incremental designers, on the other hand, take many things into account throughout implementation: How can object interactions be improved? Is there a way to reduce messaging traffic between collaborators? Are interfaces to object services simple enough or powerful enough? Can a higher information bandwidth connection be made between collaborators? Is there a way to reduce the complexity of control logic? Is polymorphism being used to advantage? Is data really being encapsulated correctly? What new classes should be created to reduce existing complexity? How might behaviors be refactored to achieve a better balance and cleaner distribution of responsibilities? Have we formed the right abstractions? What classes should be eliminated? Does the current implementation of an inheritance hierarchy facilitate or unnecessarily constrict the addition of new functionality? Are there existing interactions that could be refactored to encapsulate details or hide objects from one another? How well is the object model holding up? Are there serious flaws that demand major redesign and repair?

Incremental design involves a fundamental shift in goals, values, and process. It requires that we inject incremental design throughout implementation. To do so, we must distinguish between finishing an implementation task and completing a satisfactory design. Working code doesn't automatically signal completion. Getting the design right is a journey. That journey begins as soon as the ink has dried on system requirements. It ends when we declare an end to discovery and invention. There does come a time when we have to stop improving the design and must focus on completing our work. The tricky part is picking the right time to make that dash to the finish line. Stopping design too early means the system "evolves" rather than being "systematically planned and implemented." Stopping design too late can cause problems, too. There is always a tension between getting the design "right" and meeting the schedule. However, embracing incremental design means that change and improvements aren't viewed as threats, instead they are acknowledged and carefully factored into the development process. ▣

Rebecca Wirfs-Brock is the Director of Object Technology Services at Digitalk and co-author of DESIGNING OBJECT-ORIENTED SOFTWARE. She can be reached via email at rebecca@digitalk.com or via US mail at Digitalk, 7585 S.W. Mohawk Drive, Tualatin, OR 97062. Comments, further insights, or wild speculations are welcome.

GETTING REAL

Juanita Ewing

Don't use Arrays?

This column discusses inappropriate use of arrays and how misuse affects reusability. We will analyze several Smalltalk methods that use arrays and revise them to use classes instead of arrays. We will also show you how to search your image for methods that use arrays.

MOTIVATION

A class in Smalltalk is a specification of behavior and supporting data. Each instance contains a particular set of related data. For example, the data for an instance of Rectangle is two points. The points are related because they are both part of a rectangle: One is the origin point, and the other is the corner point.

In Smalltalk, you can also use a data structure such as an array to represent related data. Instead of the class Rectangle, you could use an array with the first element of the array being the origin point and the second element being the corner point. Which is more reusable?

First, let's examine how clients access data. Clients of the class Rectangle can send the messages origin and corner. Clients of the rectangle-as-array must access the correct element by specifying the index, and the index might not have any correlation to the values stored in the array.

Accessing the data is not the only consideration. Rectangle has specialized behavior, such as height, containsPoint:, intersects:, and expandBy:. The rectangle-as-array has no specialized behavior. For example, each client that needed the height of the rectangle-as-array would have to duplicate the code that subtracted the two y coordinates to obtain the height of the rectangle.

There are three reasons why the class is more reusable than the array:

- **Ease of Use.** Clients of the rectangle-as-array need to know arbitrary indices to obtain the data. Clients of the rectangle-as-class send messages with meaningful names.

- **Encapsulation.** The behavior of rectangle is not encapsulated with the data in the rectangle-as-array. Clients of the rectangle-as-array would need to write much more code than the clients of the rectangle-as-class in order to duplicate the behavior of rectangle. Most clients would write the same code over and over.

- **Information hiding.** The constituent data for the rectangle is accessible to all clients in the rectangle-as-array. Indeed, it must be in order for clients to the duplicate the behavior of Rectangle. But it also means the rectangle-as-array cannot change its representation without affecting all its clients.

INAPPROPRIATE USE I

Standard Smalltalk even provides us with a bad example of array usage (nobody's perfect). On page 109 of SMALLTALK-80: THE LANGUAGE AND ITS IMPLEMENTATION is the specification of a class method for Date:

Date class protocol
general inquiries

dateAndTimeNow	Answer an Array whose first element is the current date (an instance of class Date representing today's date) and whose second element is the current time (an instance of class Time representing the time right now).

Here is one possible implementation of the method:

```
Date class methods
    dateAndTimeNow
        "Answer an Array of two elements. The first element is a
        Date representing the current date and the second element
        is a Time representing the current time."
        ^ (Array new: 2)
        at: 1 put: self today;
        at: 2 put: Time now;
        yourself
```

Clients of this method must keep track of which elements are where in the array. The code to compare two date-and-time arrays looks like this (the variables now and then contain date-and-time arrays):

```
| now then oldest |
then := self oldDateAndTime.
now := Date dateAndTimeNow.
((now at: 1) >= (then at: 1) and: [(now at: 2) > (then at: 2)])
        if True: [oldest := then]
```


This kind of code is not easy to read and is likely to be duplicated in an application that manipulates time stamps.

In the dateAndTimeNow method, the array is merely a shortcut way of implementing a return of two values. The elements in the array have nothing to do with their indices. Clients have to remember which element is which. They also have to remember the algorithm for comparing date/time pairs. This kind of shortcut is not good coding practice because it does not facilitate reuse.

A better solution is to create a new class that represents an associated date and time. We will call this class TimeStamp. It would have messages for accessing its date and time, and for comparing itself with other TimeStamps. Using this new class, the dateAndTimeNow method can be rewritten:

```
Date class methods
    dateAndTimeNow
        "Answer an instance of Time Stamp containing the current
        date and the current time."
        ^TimeStamp date: self today time: Time now
```

Even better would be to eliminate the Date method and create a TimeStamp method that returns the current date and time. A TimeStamp method is better because the instance is created in the class that relates date and time. The Date class is a less desirable location because dates don't have an explicit relationship with time. Time is not referenced in other Date methods.

```
TimeStamp class methods
    now
        "Answer an instance of the receiver containing the current
        date and time."

        | current |
        current := self new.
        current date: Date today.
        current time: Time now.
        ^current
```

The client of this functionality can now write much simpler fragments of code.

```
        | now then oldest |
        then := self oldTimeStamp.
        now := TimeStamp now.
        now > then
                if True: [oldest := then].
```

INAPPROPRIATE USE II

A method from Directory provides us with another inappropriate use of an array. In this method, a collection of arrays provides detailed information about each file in a directory.

```
Directory methods
    formatted
        "Answer a collection of arrays of file information for the re-
        ceiver directory. Each array has four entries: file name, size,
        date/time and attributes."

        | answer file Entries anArray |
        file Entries := self contents.
        answer := Ordered Collection new: file Entries size.
        file Entries do: [ :each |
                anArray := Array new: 5.
                anArray
                        at: 1 put: (Directory extract FileName From: each);
                        at: 2 put: (Directory extract SizeFrom each);
                        at: 3 put: (Directory extract DateTime From: each);
                        at: 4 put: (Directory extract ResourceSize From: each);
                        at: 5 put: (Directory extract CreatorTypeFrom: each).
                answer add: anArray].
        ^ answer
```

Note that the method comment is wrong. It references an array with four entries, but the code has an array with five entries, indicating that a small change in the implementation has a big impact on clients. Users of this method must know where relevant information is stored in the array. It is impossible to tell from either the comment or the code which array element is new.

In this fragment of code, the client of Directory needs the names of files of zero length. This code must reference elements stored at arbitrary locations, and requires heavy commenting to be maintainable.

```
        | zeros |
        zeros := myDirectory formatted
                select: [:info | (info at: 2) = 0]. "size is stored at 2"
        ^zeros collect: [:info | info at: 1] "name is stored at1"
```

Related data stored in arrays is more appropriate as an instance of a class. In this example, the information stored in an array

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS MAIL PERMIT NO. 279 DENVILLE NJ

POSTAGE WILL BE PAID BY ADDRESSEE

The Smalltalk Report
Subscriber Services Dept SML
PO Box 3000
Denville NJ 07834-9821

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS MAIL PERMIT NO. 4362 SAN JOSE, CA

POSTAGE WILL BE PAIDBY THE ADDRESSE

SERVIO CORPORATION
2085 HAMILTON AVENUE
SUITE 200
SAN JOSE, CA 95125-9985

The Smalltalk Report

☞ **Provides objective & authoritative coverage on language advances, usage tips, project management advice, A&D techniques, and insightful applications.**

"If you're programming in Smalltalk, you should be reading **The Smalltalk Report**"

❑ **Yes, I would like to subscribe to The Smalltalk Report** Date_____

❑ 1 year (9 issues)
- ❑ Domestic $69.00
- ❑ Foreign $94.00

❑ 2 year (18 issues)
- ❑ Domestic $128.00
- ❑ Foreign $178.00

Method of Payment
- ❑ Check enclosed (payable to **The Smalltalk Report**)
- ❑ Bill me
- ❑ Charge my: ❑ Visa ❑ Mastercard ❑ Amex

Card No. _____
Exp. Date _____
Signature _____

Name _____
Title _____
Company _____
Address _____
City _____
State _____
Zip _____
Country _____
Phone _____

1. Which dialect of Smalltalk do you use:
- ❑ Smalltalk V
- ❑ Smalltalk-80
- ❑ Other _____

2. What is your involvement in software purchases for your department/firm:
- ❑ Recommend Need
- ❑ Specify Product
- ❑ Make Purchase
- ❑ None

3. Which operating system supports your software:
- ❑ UNIX
- ❑ DOS
- ❑ OS/2
- ❑ Windows
- ❑ Other _____

4. What is your company's primary business activity:
- ❑ Computer/Software Development.
- ❑ Manufacturing
- ❑ Financial Services
- ❑ Government/Military/Utility
- ❑ Educational/Consulting
- ❑ Other _____

5. For how long have you been using Smalltalk:
- ❑ Less than one year
- ❑ 1–3 years
- ❑ 3+ years

E3EG

A member of the Object Marketing Network

fax to 212/274-0646

SIGS PUBLICATIONS

The Smalltalk Report

The International Newsletter for Smalltalk Programmers

June 1993 Volume 2 Number 8

SMALLTALK BENCHMARKING REVISITED

By Bruce Samuelson

Contents:

When Smalltalk emerged from the Xerox PARC labs in the early 1980s, performance was a major issue. CPU speeds and memory densities were both nearly two orders of magnitude lower than in today's machines. The 1983 "green book," SMALLTALK-80, BITS OF HISTORY, WORDS OF ADVICE, included many articles with detailed performance analysis.[1] One chapter even studied the feasibility of implementing Smalltalk in hardware, namely in the Intel 432 chip. The Xerox Dorado workstation was the fastest Smalltalk machine, and implementations on chips such as the DEC VAX and Motorola 68000 did well to run at a small fraction of a Dorado.

As the decade progressed, hardware got faster at a factor of nearly 10 every five years. Efficient techniques were employed for method look-up caches and for generation-scavenging garbage collectors. By mid 1992, a midrange machine running ParcPlace Smalltalk performed several times faster than a Dorado, and a fast machine a dozen times faster. One could buy a cheap PC running either ParcPlace or Digitalk Smalltalk faster than a Dorado.

These developments raise the question of whether Smalltalk is now fast enough. Shouldn't vendors concentrate on features rather than performance? Won't hardware advances take care of any lingering problems with speed? This was, in fact, the position taken by a senior representative of one of the major Smalltalk vendors in a conversation with me last year. If Smalltalk were the only language, and if there were only one vendor, the answer might be yes. But Smalltalk implementations are not only vying with one another for prominence, they are also competing with other languages.

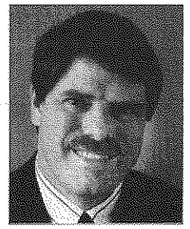
PERFORMANCE OPTIMIZATION IN OTHER LANGUAGES

One reason C++ has become so popular is that it adds object extensions to C without sacrificing much of C's efficiency. This is a frequent theme in USENET news groups such as comp.lang.c++ and is commonly cited as a reason for using C++ instead of Smalltalk. Smalltalk users cite Smalltalk's consistent use of the object paradigm, productive development environment, rich class library, flexibility, and portability (for ParcPlace's products) as reasons to choose it over C++. A language with Smalltalk's features that approaches C++'s speed would attract a larger community of users. It this possible or only a dream?

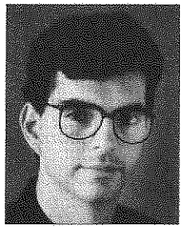
Perhaps the researchers who are most aggressively trying to demonstrate its possibility is the Self group at Stanford University. Like Smalltalk, Self is a fully dynamically typed language. It uses prototypes and delegation in place of classes and inheritance. Whereas other researchers have tried to achieve performance gains (and perhaps other benefits) by adding strong typing to Smalltalk, the Self group is seeing how far they can push the performance envelope by using various compiler optimization techniques without sacrificing type flexibility.

They have pushed the envelope quite far. An example of their results is described in an article by Craig Chambers and David Ungar in the OOPSLA 91 con-

continued on page 16...



John Pugh



Paul White

EDITORS' CORNER

Over the past 24 months, we have often discussed Smalltalk's move into the business world. Both Digitalk and ParcPlace have spent a significant effort to not simply improve their existing products, but instead to change their products to position Smalltalk as the best development tool for large organizations across all industries. To this end, both PARTS and VisualWorks represent the next generation of products for their respective vendors, which attempt to make Smalltalk more accessible to the mass development market—and newer Smalltalk vendors are sure to arrive. Easel's Enfin product is already having an impact on the object-oriented market that is likely to grow as time goes on.

Recently, we have noted that Smalltalk is being talked about in arenas that would not have been dreamed of before. One such place was a recent column in the April 19th issue of **Business Week** in which Smalltalk is described as being an extremely successful development tool for many corporations including American Airlines, JP Morgan, and Citicorp, and the list of these companies keeps growing. Reports such as these can be used as fodder for those of you who are still fighting to justify Smalltalk to your management.

In our feature article this month, Bruce Samuelson offers some benchmarks he has performed for the various dialects of Smalltalk. This is a new arena for THE SMALLTALK REPORT, and we believe efficiency is an issue that many of you face "in the trenches." Bruce has been very active in recent months on Internet discussing this topic, and has invested a great deal of time in preparing this study. More important than the raw numbers he presents, he has many insightful comments concerning the implementation strategies of both ParcPlace and Digitalk. While not endorsing the numbers presented by Bruce, we strongly believe these types of studies are crucial to the further mainstreaming of Smalltalk.

The debate over whether to use accessor methods has raged in the Smalltalk community since "the beginning of time." As Kent Beck points out in his column this month, this one question has probably been debated more vehemently in Smalltalk labs than any other style issue. In our own shop, the question of the appropriate use of accessors has been argued so much that it is now considered a taboo subject. We believe Kent has put this debate in the right context, especially the comment that programmers will "do anything, given enough stress," and suggest anyone responsible for the integrity of their corporate libraries give these arguments attention.

This month we have two columns that address the issue of GUI development using Smalltalk. First, Greg Hendley and Eric Smith return this month with their GUI column, getting you started with integrating VisualWorks with Microsoft's Help facility. Second, Eric Clayberg and S. Sridhar take a first look at WindowBuilder Pro, the next generation of the well-known WindowBuilder product originally released by Cooper and Peters.

Alan Knight's look at comp.lang.smalltalk takes him into a review of the implementation of sets and dictionaries. In doing so, he studies how well (or not well) abstracted the implementation of these reusable data types is and some suggestions for improving them. Also this month, Wayne Beaton describes an implementation of a mechanism for storing and managing DLLs for Smalltalk/V for windows.

Enjoy the issue!

John Pugh Paul White

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.

Mailed First Class. Subscription rates 1 year, (9 issues) domestic, $65. Foreign and Canada, $90, Single copy price, $8.00. POSTMASTER: Send address changes and subscription orders to: THE SMALLTALK REPORT, Subscriber Services, Dept. SML, P.O. Box 3000, Denville, NJ 07834. Submit articles to the Editors at 91 Second Avenue, Ottawa, Ontario K1S 2H4, Canada. For service on current subscriptions call 800.783.4903.
Printed in the United States.

PRODUCT ANNOUNCEMENTS

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

ICONIX Software Engineering's **ObjectModeler** now supports Smalltalk. ObjectModeler is an OOA/OOD/OOP module. This recent addition was made in response to the developing trend in the object-oriented market that more and more COBOL and IS shops are moving into Smalltalk while technical shops continue to move into C++.

ICONIX ObjectModeler already supports C++ and SQL development and the company believes that the addition of Smalltalk will be of particular interest within the IS market. ObjectModeler users already have the ability to attach text files to any symbol on a Rumbaugh, Coad/Yourdon, or Booch diagram within ObjectModeler. In the same way that C++ and SQL templates are used to link source code to diagrams, they can now pick from 9 menus containing over 270 Smalltalk language constructs.

ICONIX Software Engineering, 2800 28th St., Suite 320,
Santa Monica, CA, 310.458.0092 (v), 310.396.3454

WindowBooster is a simple and powerful utility that optimizes the opening of windows and dialog boxes programmed using Digitalk's Smalltalk/V. WindowBooster significantly improves the overall speed of any application. The product is easy to install, transparent to the user, and compatible with products such as WindowBuilder. The product is available for Windows and OS/2 and includes complete source code.

Tau Ceti, 1801 Avenue of the Stars, Suite 404, Los Angeles,
CA 90067-5906, 310.556.9723 (v), 310.556.9725

Tensegrity is an object-oriented database system for Smalltalk. Using Tensegrity, Smalltalk developers can create single-user or multi-user network applications without changing code. The product provides transparent object persistence, advanced transactional capabilities, two-phase commit, distributed garbage collection, and exceptional speed. Because the product is network-independent and requires no dedicated database server, the company anticipates that it will have great appeal to developers of workgroup applications.

Polymorphic Software, 1091 Industrial Rd., Suite 220, San Carlos,
CA 94070, 415.592.6301 (v), 415.592.6302 (f)

continued from page 8

the outside world. Messages should present the services an object is willing to provide. Using them to give an abstract view of storage turns those implementation decisions into yet more services. Revealing implementation is exactly what encapsulation is supposed to avoid.

"Just make the accessors private." That's the common solution, but there are two reasons why this isn't a sufficient solution. First, anyone can invoke any method (and will, given enough stress). There is currently no way to make truly private methods that cannot be used outside the class. Digitalk and ParcPlace are both working on this problem. More seriously, programmers are notoriously bad at deciding what should be private and what should be public. How many times have you found "just the right method," only to find it marked private? If you use it, you are faced with the possibility that it may go away in the next release. If you don't, you have to violate the encapsulation of the object to do the computation yourself, and you have to be prepared for that computation to break in the next release.

The argument against automatically using accessors rests on the assumption that inheritance is less important than encapsulation. Rick DeNatale of IBM argues that inheritance should be kept "in the family." Anytime you inherit from a class you don't own, your code is subject to unanticipated breakage much more than if you merely refer to an object. If you want to use inheritance, do it only between classes whose change you control. While this may not result in the most elegant solution, it will save you headaches in the long run.

Using this model, you can access variables directly. If you want to make a subclass that needs to access a variable through a message, you use the programming environment to quickly change "x := ..." into "self x: ..." and "x ..." into "self x ...". Encapsulation is retained, and the cost of changing your decision is minimal. If you don't own the superclass or the subclass, you can't do this, as it would involve making changes in code you can't control.

CONCLUSION

Aesthetics does not provide a compelling argument one way or the other. There's a giddy feeling when you make a subclass the original programmer never anticipated, but only need to make a few changes to make it work. On the other hand, there is satisfaction in thinking you finally have to reveal a variable, only to discover that by recasting the problem you can improve both sender and receiver.

Regardless of how you choose to program, you are faced with the hard choice of deciding which variables should be reflected as messages. Pushing behavior out into objects rather than just getting information from them and making decisions yourself is one of the most difficult, but most rewarding, jobs when programming objects. Making an accessing method public should be done only when you can prove to yourself that there is no way for the object to do the job itself. Making a setting method public requires even more soul-searching, since it gives up even more of an object's sovereignty.

Either way, you accept a discipline not supported by the language. If you choose to use accessors, you and everyone who uses your code must swear an oath never to send messages that invoke methods marked private in the receiver. You also must be wary of using the accessor from outside the object when you really need to add more services to the receiver. If you do not use accessors, you accept the burden of refactoring classes, either making an abstract class or at least adding accessors, should a later inheritance decision make it necessary.

> "Programmers are notoriously bad at deciding what should be private and what should be public."

Whichever style you choose, make sure it pervades your team's development. Einstein is reputed to have said, "You can be consistent or inconsistent, but don't be both." The same simplifying assumptions should hold throughout all of your code.

If you use accessors, make them all private at first. Only make them public if you must, and struggle to discover a less centralized solution first. Don't assume that because you access variables through messages you have made all of the abstraction decisions you'll have to make. Using an accessor, internally or externally, should alert you that there may be missing behavior.

If you use variables directly, be prepared to recant your decision when the time comes. If what you thought was state is really a service, make the change everywhere. Don't have external users getting a variable's value through a method and internal users accessing it directly.

So, what's The Answer? In my own code, I change state into service (define an accessing or setting method) only when I am convinced it is necessary. Otherwise, my classes access their variables directly. I think inheritance is overrated. Providing the right set of services has more bearing on the success of a design. There are plenty of successful, experienced folks who would call me a reactionary hick for this (and worse things, for other reasons). Try some code each way and decide for yourself which style you find more comfortable. That's the only right answer. ▣

Kent Beck has been discovering Smalltalk idioms for eight years at Tektronix, Apple Computer, and MasPar Computer. His is 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, by phone at 408.338.3666, or on CompuServe at 70761,1216.

Using Windows Resource DLLs from Smalltalk/V

Wayne Beaton

Microsoft provides a handy mechanism for Windows-compliant applications to store resources in Dynamic Link Libraries (DLL). While an extensive tool set exists to access resources stored in DLLs, seasoned Smalltalk programmers are a little spoiled and generally hope to avoid contact with operating system details. I have implemented a Windows resource DLL manager in Smalltalk to protect hardworking problem solvers from the semantics of dealing with Windows directly. The resources of primary interest are bitmaps, icons, and cursors; I have left room, however, for expansion to include resources such as string tables and perhaps programmer-defined resources.

A resource dynamic link library can be constructed reasonably easily—provided you have a resource compiler and a lot of time to figure out how to use it. Fortunately, Digitalk provides a resource DLL for free: the file vwsignon.dll contains the dialog that Smalltalk displays as it loads itself during runtime. A copy of this file, placed in the working directory of the image, can be easily modified by a resource editor.

All the Windows functions required to access DLLs, which are detailed in The Microsoft Windows Software Development Kit (SDK) manuals, have hooks in the base Smalltalk/V image. Also in the base image is the class DynamicLinkLibrary, which provides an abstract representation of a DLL. Equipped with this class and the battery of existing methods, all that is really required is management of the resources.

The class WindowsResourceManager has been developed to manage resource DLLs. As an instance is created, it is provided with the name of the DLL file whose resources it represents. The instance will automatically open the DLL when required and will automatically close it when the image is either saved or exited. The programmer need only ask the instance for a particular resource by type and name. The methods bitmapAt:ifAbsent:, cursorAt:ifAbsent: and iconAt:ifAbsent answer the named bitmap, cursor, or icon, respectively. The first parameter is a case-independent string containing the name of the resource; the second is a block to evaluate if the resource cannot be successfully accessed.

As each resource is loaded, it is cached to prevent the same resource from monopolizing system resources. The Windows Graphics Device Interface (GDI), for example, allocates a special handle for bitmaps. As only a relatively small number of these handles are available, frugal use will allow many bitmaps to be used frequently. Caching also relieves the programmer of the responsibility of releasing the DLL resources; all cached resources are released when a WindowsResourceManager closes itself.

The provided example methods show how an instance of WindowsResourceManager might be used. In Listing 1, an instance is created using the message

```
WindowsResourceManager class>>onDLLNamed:
```

and stored in a global variable. The instance is then asked for a bitmap with the message

```
WindowsResourceManager>>bitmapAt:ifAbsent:
```

Inspection of the method

```
WindowsResourceManager>>bitmapAt:ifAbsent:
```

reveals that the receiver is first opened. Then the cache is inspected to see if a bitmap already exists with the provided name. That failing, Windows is asked to find the bitmap. If no bitmap exists, the ifAbsent block is evaluated.

When an instance of WindowsResourceManager is asked to open, it first checks to see if it is already open. If it is not, it attempts to open the DLL it is to access and remembers it. After the DLL has opened, it tells Smalltalk to notify it on exit. The method SystemDictionary>>notifyAtExit: ensures that the instance will be notified with the message WindowsResourceManager>>exit when Smalltalk attempts to exit gracefully.

The method WindowsResourceManager>>exit simply closes the instance, releasing the resources which have been loaded, closing the DLL and removing itself from notification with the method SystemDictionary>>removeExitObject:.

When the image is saved, all classes are sent the message aboutToSaveImage. The class WindowsResourceManager reroutes this message to all of its instances. Each instance directs itself to close when the image is about to be saved. Long-term references to resources should be avoided: Accessing resources exclusively through the WindowsResourceManager will avoid embarassment when they are automatically released as the image is saved.

The code that I have included provides all the necessary equipment to effortlessly access bitmaps, cursors and icons from a DLL. As always, I am open to any suggestions as to how this may be extended, or modified for efficiency. ▣

Wayne Beaton is a senior member of the Technical Staff at the Object People. He likes to think of objects as having personality as well as behavior. He can be contacted at the Object People at 613.225.8812 (v) or 613.225.5943 (f).

and ST/V, if I had used more plain vanilla classes, they could have had a wider reach.

Slopstones is so low-level that many of its individual tests may get completely optimized away by the compiler. I knew this wouldn't happen with current Smalltalk compilers, but it did happen when Urs Hölzle compiled it under Self.

The Smopstone test for sorting a set of strings was subtly flawed. The raw material for the sort was different for ST80 and ST/V because the ST80 Set enumerates an instance from low index to high, while ST/V-DOS enumerates from high to low. Moreover, the sets being sorted are hashed differently resulting in a different ordering of their elements. If I had sorted the original array of strings rather than the derived set of strings, this flaw would be removed. The result of doing this is to slow down the ST80 sort speed by 5–10% while leaving the ST/V sort speed virtually unchanged. In other words, ST/V wins the sort test by 5–10% more than in the Smopstone chart.

CAN SMALLTALK PERFORMANCE BE FURTHER OPTIMIZED?

The benchmarks in this article show that there are areas in which ST80 excels over ST/V and others in which ST/V excels. This suggests that both ParcPlace and Digitalk could wring out better performance by conventional means. As for more exotic optimizations, the Self researchers claim the answer to the question is most definitely *yes,* both in their publications and in private conversations. Vendor representatives are less convinced. I have only talked with ParcPlace, but my impression is that they either feel it is not technically feasible to achieve Self performance in a commercially viable way (e.g., without requiring 64MB machines), or it would be too expensive for them to do it, or their customers do not regard it as a priority.

Last May there was a flurry of discussion in comp.lang.smalltalk on Smalltalk efficiency. One thread focused on the ParcPlace virtual machine, and in particular, on whether using register windows in native machine code on Sun Sparc platforms would speed it up much. A second thread focused on whether Self optimization techniques could be applied profitably to Smalltalk. After considerable discussion, Peter Deutsch made a summary statement for both threads. He used to be with ParcPlace and has considerable experience in implementing and optimizing Smalltalk. Regarding the second thread, he expressed the following private opinion (his views do not necessarily reflect those of his employer):

> As for the comparison [of Smalltalk] against Self, the Self authors acknowledge that the factor of 5 [improvement of Self over Smalltalk] is only achievable under some circumstances. I do think it would be exciting to apply the Self compilation ideas to Smalltalk, and doing this could well produce dramatic performance improvements (on all platforms), but this would require wholesale redesign of most of the platform-independent code (other than the memory manager) in the [ST80] runtime support system. The optimizing compilation experiments I did at ParcPlace were based on an alternative approach that would not have required such substantial changes to the [ST80] virtual machine, but might have required type declarations (or at least type hints) provided by the user (or a type inference system).

I don't know whether ParcPlace has continued their experiments or whether Digitalk has any active projects to push toward Self's performance. It is interesting that two of the prime movers, Peter Deutsch (Smalltalk) and David Ungar (Self) have moved respectively from ParcPlace Systems and Stanford University to Sun Microsystems. I wonder what Sun has up its sleeve?

In conclusion I would urge you to let your vendor know if performance optimization is important to you. Report serious bottlenecks to them. I have found ParcPlace to be quite responsive in correcting them.

COMPILING AND RUNNING THE BENCHMARKS

The benchmarks require floating point hardware or emulation software. They compile and run without difficulty on all the versions of ST80 and ST/V for which they have been tested. At least two Smopstone benchmarks, fractonacci and rectangle intersection, won't run under GNU Smalltalk because it lacks Fraction and Rectangle classes.

It is a good idea to file the code into a clean image and do a garbage collect before running it if possible. The individual times will fluctuate somewhat, but the geometric mean is pretty stable. You can reduce fluctuations by running more iterations (the n variable in the execute method). Doing so for ST/V-DOS may crash it though.

Be sure to run ST/V-DOS benchmarks under native DOS. For example, Smopstones in a full screen DOS shell under Windows only runs at 62% of its speed under native DOS.

Mail the results to me or post them to comp.lang.smalltalk. If you want to try the Self performance suites, contact self-request@self.stanford.edu. Or ftp from the directory benchmarks/st80-2.4.

SOURCE CODE

You may ftp the source code from the public domain Smalltalk archives at the University of Illinois (st.cs.uiuc.edu 128.174.241.10) or University of Manchester (mushroom.cs.man.ac.uk 130.88.13.70). ▣

REFERENCES

1. G. Krasner. Smalltalk-80, Bits of History, Words of Advice, Addison-Wesley, Reading, MA, 1983.
2. Chambers, C., and D. Ungar Making pure object-oriented languages practical, OOPSLA 91 Conference Proceedings; also published as SIGPLAN Notices 26.11, November 1991.

Bruce Samuelson uses ParcPlace Smalltalk for linguistic applications at the University of Texas at Arlington and with the Summer Institute of Linguistics. Bruce can be reached via internet at bruce@utafll.uta.edu (uta-eff ell ell).

ST/V-Windows comes off rather poorly. I don't know whether this is because Digitalk hasn't optimized it as much as their OS/2 versions or because the only test ran it under Windows which itself ran under OS/2. Perhaps it would run faster under native Windows. A recent COMPUTERWORLD article says that Digitalk is beta testing a new Windows version based on The Win32s 32-bit interface. It yields "a big performance boost" and is expected to be ready in July.

You definitely want to run ST/V-DOS under native DOS rather than in a DOS shell under Windows. In the latter it runs at only 62% of native capacity.

Although ST/V-Windows beat ST/V-DOS by 0.167 to 0.070 on Slopstones for a 486/33, they came in nearly tied on Smopstones. Moreover, the individual Smopstone tests, which are not given in this article, were quite close for the two versions. Since the DOS version I tested was 1987 vintage or earlier (its file dates were 1987), this suggests that Digitalk's Windows version is in need of a performance tune-up. I don't understand the divergence between low- and medium-level results.

The Macintosh results for ST/V aren't too bad if you omit streams and sets from Smopstones. I haven't included the individual runs, and I don't have a Mac version. I think the same theory as outlined earlier applies; namely, ST/V-Mac must be really bad on mixed mode integer-float arithmetic, which streams use, and absolutely terrible on string hashing, which set formation uses. I've heard that a new Mac version may be shipping by the time this article is published.

Ideally, the tests comparing ParcPlace to Digitalk should be made on the same machine. I am assuming that the 486/33 on which I tested ParcPlace is about equal to the 486/33 on which Marten Feldtmann tested Digitalk. Similar comments can be made about the Macs. Although my machine benchmarks faster than Marten's on ParcPlace's Dorado benchmarks, I think this is due to differences in our video cards.

There are several means one could report. Three popular ones are:

- Arithmetic mean = (x1+x2...+xn)/n
- Harmonic mean = n/((1/x1)+(1/x2)...+(1/xn))
- Geometric mean = (x1*x2...*xn)**(1/n)

I chose geometric mean because it has the best scaling properties and it is the least sensitive to one number being particularly low or high. The ParcPlace Dorado benchmarks use harmonic mean. When I first posted the benchmarks to comp.lang.smalltalk, I used geometric mean, but erroneously called it *harmonic*. Urs Hölzle corrected me.

The benchmarks have several shortcomings, some of which were pointed out by people posting to comp.lang.smalltalk. There should be a lot more than seven tests in each suite. They concentrate on too few areas of Smalltalk and omit many of the diverse capabilities of its class library. With so few tests, they could be sensitive to one particularly weak link in an implementation. Examples of such links we probably encountered were a bad string hash function for ST/V (especially for Mac), weak floating point performance for ST/V, poor string compare or sort algorithm for ST80, and possibly poor recursion or poor performance of nonclean blocks in ST80 (e.g., fibonacci and fractonacci, and especially Marten Feldtmann's while loop.).

Two low-level tests I wish I had included in Slopstones would have been to test direct method dispatch efficiency with

```
Object new yourself; yourself; yourself...
```

and then to test inherited dispatch with something like

```
Dictionary new yourself; yourself; yourself...
```

This would have determined the absolute maximum number of method dispatches that can be performed per second. In an informal test, direct dispatch with ST80 ran at the same speed as integer addition. On a 486/33, this means you get a maximum of 6.6 million dispatches per second. Most machine language instructions probably run in one clock cycle, yielding 33 million machine language operations per second (MIPS). The actual mips rating of a 486/33 is perhaps half or two third this, but is still much higher than its MDPS (million dispatches per second) rating.

Other additions to Slopstones could be using arguments when evaluating blocks and performing selectors. One could imagine many more, too.

I received some suggested additions to Smopstones by email after I had already frozen them. I also should have included some of the benchmarks already developed by the Self group. Richards was donated to them by Peter Deutsch, formerly of Parc Place.

The benchmarks are not at a level high enough to test actual applications. Slopstones, in particular, is hardly a predictor of real-life performance. However, by comparing ST80 and ST/V on low-level and medium-level operations, the style of benchmark we have written does shine the spotlight on operations that need to be optimized. If Slopstones and Smopstones were made more comprehensive, this could help the vendors find areas in which their performance is not competitive.

The benchmarks do not test the speed of user interactions such as opening windows or scrolling lists. These consume a lot of a user's time in practice. Nor do they test how quickly Smalltalk accesses disk files, which can be important in some applications. For example, I can read an ASCII file on my 486/33 machine running ParcPlace at only 40K bytes per second when doing high-level access with contentsOfEntireFile. Although much higher rates are achievable with IOAccessors (in the 1MB range), this is low-level and inconvenient.

Although the benchmarks are portable between ST80 and ST/V, they are less portable to other languages. Urs Hölzle ported Slopstones to Self easily enough, but couldn't easily port Smopstones because Self lacks streams and fractions. A user of GNU Smalltalk couldn't port Smopstones because GNU lacks rectangles and fractions. Although I hadn't anticipated the benchmarks being run for any languages except ST80

Listing 1.

```
Object subclass: #WindowsResourceManager
    instanceVariableNames:
        'fileName dll cachedResources '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'DLL'!

!WindowsResourceManager class methods

aboutToSaveImage
    "When the image is about to be saved,
    inform any of my instances."
    "(WindowsResourceManager aboutToSaveImage)"
    self allInstancesPrim do: [:each | each aboutToSaveImage]! !

!WindowsResourceManager class methods

example1
    "Answer the icon named 'Balloon' in the dll named
    'vwsignon.dll'."
    "(WindowsResourceManager example1)" | resources |
    resources := WindowsResourceManager on: 'vwsignon.dll'.
    ^resources iconAt: 'Balloon'! !

!WindowsResourceManager class methods

onDLLNamed: aString
    "Answer an instance of myself for use
    with the DLL named aString."
    ^self new fileName: aString! !

!WindowsResourceManager methods

aboutToSaveImage
    "When the image is about to be saved, close myself so that next
    time the image is opened, I open in a clean state."
    self close! !

!WindowsResourceManager methods

bitmapAt: aString
    "Answer the bitmap named aString."
    ^self bitmapAt: aString ifAbsent: [self error: 'No such bitmap.']!

bitmapAt: aString ifAbsent: block
    "Answer the bitmap named aString. If no such bitmap exists,
    then evaluate block (with no parameters)."
    | key |
    self open.
    key := Array with: Bitmap with: aString asUpperCase.
    ^self cachedResources
        at: key
        ifAbsent: [
            self cachedResources
                at: key
                put: (self buildBitmapNamed:
                    aString ifAbsent: [^block value])]!

buildBitmapNamed: aString ifAbsent: block
    "Private - Answer the bitmap named aString."
    | handle |
    handle := UserLibrary
        loadBitmap: self dll asParameter
        name: aString asParameter.

    handle = 0 ifTrue: [^block value].

    ^Bitmap fromHandle: (WinHandle fromInteger: handle)! !

!WindowsResourceManager methods

buildCursorNamed: aString ifAbsent: block
    "Private - Answer the cursor named aString."
    | handle |
    handle := UserLibrary
        loadCursor: self dll asParameter
        name: aString asParameter.

    handle = 0 ifTrue: [^block value].

    ^CursorManager fromHandle: (WinHandle fromInteger: handle)!

cursorAt: aString
    "Answer the cursor named aString."
    ^self cursorAt: aString ifAbsent: [self error: 'No such cursor.']!

cursorAt: aString ifAbsent: block
    "Answer the bitmap named aString. If no such bitmap exists,
    then evaluate block (with no parameters)."
    | key |
    self open.
    key := Array with: CursorManager with: aString asUpperCase.
    ^self cachedResources
        at: key
        ifAbsent: [
            self cachedResources
                at: key
                put: (self buildCursorNamed:
                    aString ifAbsent: [^block value])]! !

!WindowsResourceManager methods

buildIconNamed: aString ifAbsent: block
    "Private - Answer the icon named aString."
    | handle |
    handle := UserLibrary
        loadIcon: self dll asParameter
        name: aString asParameter.

    handle = 0 ifTrue: [^block value].

    ^Icon fromHandle: (WinHandle fromInteger: handle)!

iconAt: aString
    "Answer the icon named aString."
    ^self iconAt: aString ifAbsent: [self error: 'No such icon.']!
```

continued on page 6

```
iconAt: aString ifAbsent: block
    "Answer the bitmap named aString.
    If no such bitmap exists,
    then evaluate block (with no parameters)."
    | key |
    self open.
    key := Array with: Icon with: aString asUpperCase.
    ^self cachedResources
        at: key
        ifAbsent: [
            self cachedResources
                at: key
                put: (self buildIconNamed: aString ifAbsent:
                    [^block value])]! !

!WindowsResourceManager methods

cachedResources
    "Private - Answer my collection of cached resources."
    ^cachedResources!

cachedResources: aDictionary
    "Private - Set my collection of cached resources."
    cachedResources := aDictionary!

initializeCachedResources
    "Private - Initialize my resources cache."
    self cachedResources: Dictionary new!

releaseCachedResources
    "Private - Explicity release the cached
    resources to free up system resources."
    self cachedResources do: [:each |
        each release]! !

!WindowsResourceManager methods

close
    "Close myself. If I am not open then do nothing.
    Otherwise, release my cache and free my DLL.
    Set my DLL to nil so that I know I'm closed.
    Remove myself from notification at exit."
    self isOpen
        ifTrue: [
            self
                releaseCachedResources;
                initializeCachedResources.
            self dll free.
            self dll: nil.
            Smalltalk removeExitObject: self]!

exit
    "Force myself to close before exiting
    if I have not already done so."
    self close!

fileName
    "Answer my file name."
    ^fileName!

fileName: aString
    "Set my file name."
    fileName := aString!

isOpen
    "Answer whether or not I am open."
    ^self dll notNil!

open
    "Open myself with the resources in my file.
    Tell smalltalk to notify me before
    exiting (or saving the image), so that I
    can clean up.  If I am already open, then
    do nothing."
    self isOpen ifFalse: [
        self
            initializeCachedResources;
            dll: self openDLL.
            Smalltalk notifyAtExit: self]! !

!WindowsResourceManager methods

dll
    "Private - Answer the DLL which actually contains my resources."
    ^dll!

dll: aDynamicLinkLibrary
    "Private - Set the DLL which
    actually contains my resources."
    dll := aDynamicLinkLibrary!

openDLL
    "Private - Answer an instance of DynamicLinkLibrary
    opened on my file name."
    ^DynamicLinkLibrary open: self fileName! !
WindowsResourceManager comment: ''!
```


- Adding integers
- Adding floats
- Accessing a character in a string
- Creating an object
- Copying an object
- Performing a unary selector
- Evaluating a block without arguments

Each test is repeated many times inside a block. For example, integer addition looks like [1+1+1+1... many times]. The block, in turn, is evaluated many times.

Table 4. Benchmark results of 32-bit implementation.

slopstones (low-level)	benchmark (med level)	smopstones	benchmark
1.09	add integers	1.46	generate fractonaccis
0.53	add floats	1.09	generate primes
0.56	access strings	0.14	generate and parse streams
0.62	create objects	0.68	generate strings
0.45	copy objects	0.30	form sets
0.11	perform selectors	2.19	sort strings
0.12	evaluate blocks	0.86	intersect rectangles
0.41	geometric mean	0.71	geometric mean

SMOPSTONES

The seven medium-level tests are:

- Generating fractonaccis (like fibonacci, but using fractions)
- Generating prime numbers
- Generating and parsing streams
- Generating and manipulating strings
- Forming a set of strings
- Sorting this set
- Recursively creating sets of overlapping rectangles

Each test is repeated once using fixed values for its parameters. It can be repeated more times if necessary for fast machines. I used fractonacci rather than fibonacci because fibonacci runs were either too fast or generated 32-bit integers. Fractonacci fit within the constraints imposed by my goals.

ANALYSIS OF THE RESULTS

Digitalk didn't beat ParcPlace after all, at least in these benchmarks. The fastest version of ST/V for Intel machines ran at 41% of ST80 for the low-level tests and 71% for the medium-level tests. However, the numbers in the chart are the geometric mean of seven individual tests (x1*x2*...*x7)**(1/7). Digitalk did beat ParcPlace on some of the tests.

The results of comparing ST/V-OS/2 (32-bit) relative to ST80-Windows (32-bit) are presented in Table 4. Numbers greater than one mean ST/V is faster. Marten Feldtmann did these ST/V runs and I did the ST80 runs.

Table 4 suggests the two vendors have optimized different parts of their systems. For example, on the low-level tests, the two versions add integers at about the same speed, but Digitalk is quite inefficient at performing selectors and evaluating blocks without arguments. ParcPlace is consistently better on the remaining tests by a factor of two.

For the medium-level tests, Digitalk whips ParcPlace on sorting. Perhaps this is because Digitalk's string compare is better or perhaps they are using a better sorting algorithm. I haven't checked. Digitalk also beats ParcPlace on fractonaccis with the same margin they won on Marten Feldtmann's fibonacci test. I think this is because Digitalk is faster on tight, recursive block or method calls and—or—because of the performance penalties ParcPlace pays for full blocks and copying blocks versus clean blocks, a distinction I doubt Digitalk makes. However, Digitalk got clobbered on the stream tests, possibly because it is slow at mixed mode arithmetic between integers and floats. And it fared badly on set formation, probably because its hashing algorithm for strings is less effective than the sophisticated one used by ParcPlace, especially for ST/V Mac 1.2.

If we omit these stream, set, and sort tests from Smopstones, the 32-bit version of ST/V for OS/2 comes in at 98% of the geometric mean of ParcPlace's 32-bit version for Windows—a dead heat.

There is a wide divergence between the low- and medium-level results. ST/V-OS/2 rose from 0.41 on Slopstones to 0.71 on Smopstones. The most dramatic rise was for ST/V-DOS. It rose from 0.070 to 0.261 on a 486/33 and from 0.002 to 0.008 on an 8088/4.77. Perhaps ST/V-DOS bogs down more on Slopstone garbage collection than on Smopstones. This is pure speculation. The general advantage that ST80 has over ST/V in low-level tests relative to medium-level ones may be caused by the performance penalties that ST80 pays for distinguishing between clean, copying, and full blocks. I may have written the medium-level tests to be more susceptible to this distinction. More tests would be needed to verify this hypothesis. Recall above how I sped up the while loop for ST80 in Marten Feldtmann's test by converting a full block to a clean one.

ParcPlace would have come off slightly better if the tests were not constrained by portability. Some of the code could have been shortened by using ParcPlace's larger class library. More significantly, some of the variables that are declared as method temporaries could have been declared as block temporaries, thus converting some dirty blocks to clean ones. This would have left Slopstones unaffected, but would have improved ParcPlace's relative Smopstone performance by 2.5% on average, with the biggest gain coming in the intersecting rectangles (19%).

ParcPlace's syntax for declaring block temporary variables is shown below. ST/V-DOS does not support it. I don't know whether newer Digitalk versions do.

```
[:arg1 :arg2 |
| temp1 temp2 temp3 |
statements]
```

Table 3. Slopstone and smopstone results.

Vendor	Version	GUI	OS	Brand	CPU	flPt	MHz	Extrn cache, KB	RAM MB	Slopstone (low),*	Smopstone (med)*
PPS	VW 1.0	OpW3.0	SunOS 4.1.3	Sun SS/10-30	SPARC	int	36	0+	32	.905	1.932+
PPS	VW 1.0	—	HP/UX 2.7	HP 720	PA	intrn	50?	—	32	1.498	1.673
PPS	VW 1.0	Win3.1	DOS 5.0	Amax none	486DX	intrn	33	256	16	1.0	1.0
PPS	80 4.0—	—	Sun	SS/2	SPARC	—	40	64	64	1.137	0.995
Dig	V 2.0	PM	OS/2 2.01b	clone	486DX	intrn	33	256	16	0.411	0.982‡‡
PPS	80 4.0	Win3.1	DOS 5.0	Amax none	486DX	intrn	33	256	16	0.995	0.973
Dig	V 2.0	PM	OS/22 2.01b	clone	486DX	intrn	33	256	16	0.411	0.71
PPS	VW 1.0	Mac	MacOS 7.01	MacQuadra700	68040	intrn	25	0?	20	0.525	0.572
Dig	V 1.4	PM	OS/2 1.4	clone	486DX	intrn	33	256	16	0.236	0.470
Dig	V 1.2	Mac	MacOS	Mac accel***	68040	intrn	25	—	—	0.137	0.344‡‡
Dig	V 2.0c	—	DOS 5.0	Amax none	486DX	intrn	33	256	16	0.070	0.261
Dig	V 2.0	WinOS2	OS/2 2.01b	clone	486DX	intrn	33	256	16 0	167	0.25
Dig	V 1.2	Mac	MacOS 7.0.1	Mac II ci	68030	68882	25	32?	16	0.078	0.191‡‡
Dig	V	Mac	MacOS 7.01	Mac II ci	68030	68882	33?	0?	5/6	0.072	0.184‡‡
PPS	VW 1.0	Mac	MacOS 7.01	Mac IIci	68030	68882	25	0?	16	0.174	0.180
Dig	V 1.2	Mac	MacOS	Mac accel***	68040	intrn	25	—	—	0.137	0.131
PPS	80 4.0	OpW2.0	SunOS 4.1	Sun 3/50	68020	68881	16	0	12	0.114	0.107
PPS	80 2.5	SunVw	SunOS 4.1	Sun 3/50	68020	68881	16	0	12	0.067	0.102**
Dig	V286 1.2	none	DOS 5.0	OPTI	386DX	none	25	0	4	none	0.096
Dig	V 1.2	Mac	MacOS 7.0.1	Mac IIci	68030	68882	25	32?	16	0.078	0.072
Dig	V	Mac	MacOS 7.01	Mac IIci	68030	68882	33?	0?	5/6††	0.072	0.069
Dig	V	Mac	MacOS 7.01	Mac PB100	68000	none	16	0	3/4††	0.020	0.051‡‡
Dig	V	Mac	MacOS 7.01	Mac PB100	68000	none	16	0	3/4††	0.020	0.019
Dig	V 2.0	none	DOS 3.3	clone XT	8088	none‡	5	0	640K	0.002‡	0.008‡
Dig	V	Mac	MacOS 7.01	Mac IIsi	68030	none	25	0?	3/4††	0.044	none

* *Results are normalized to one for VisualWorks 1.0 on my 486/33.*

! *SS/10-30 has 36K internal cache. 80486 and 68040 (and I think SS/2) have 8K.*

‡ *Floating point performance was extrapolated assuming an 8087.*

** *Smopstones didn't include set formation benchmark—string hash inadequate.*

†† *x/y means Mac allocated x MB to Smalltalk out of y MB total.*

‡‡ *Smopstones excluding the two worst cases (stream, set) and the best case (sorting). The stream and set results were bad for ST/V Intel and atrocious for ST/V Mac, probably because of weak implementations of mixed integer and float arithmetic (used in streams) and string hash (used in forming sets).*

*** *This machine was a Mac IIci with a 25 MHz 68040 Radius Rocket accelerator.*

Note: The entries are sorted by Smopstones (last column). Higher numbers in the last two columns mean greater speed. Eight people contributed these results. For the 486, I did the PPS runs and ST/V-DOS run. Marten Feldtmann did the remaining ST/V 486 runs.

Stones) and *smopstones* (Smalltalk Medium-level OPeration Stones), and the results are summarized in Table 3.

I wanted to avoid any tests that stressed the disk or video systems. Although these are important in real applications, modern caching disk controllers and video coprocessors make it hard to make objective cross-platform comparisons. Also, portability is difficult to achieve between ST80 and ST/V in video tests.

SLOPSTONES

The seven low-level tests are:

SMALLTALK IDIOMS

Kent Beck

To accessor or not to accessor?

A debate has been raging on both CompuServe and the Internet lately about the use and abuse of accessing methods for getting and setting the values of instance variables. Since this is the closest thing I've seen to a religious war in a while, I thought I'd weigh in, not with the definitive answer, but with at least a summary of the issues and arguments on both sides. As with most, uh, *discussions* generating lots of heat, the position anyone takes has more to do with attitude and experience than with objective truth.

First, a little background. The classic accessor method comes in two flavors, one for getting the value of an instance variable:

```
Point>>x
    ^x
```

and one for setting an instance variable:

```
Point>>x: aNumber
    x := aNumber
```

Accessing methods are also used to do lazy initialization, or as caches for frequently computed values:

```
View>>controller
    ^controller ifNil: [controller := self getController]
```

ACCESSORS

When I was at Tektronix, Allen Wirfs-Brock (now a Digitalk dude) wrote (or at least discussed writing—it was a while ago) a think piece called "Instance variables considered harmful." His position was that direct reference to instance variables limits inheritance by fixing storage decisions in the superclass that can't be changed in a subclass. His solution was to force all accesses to instance variables to go through a method. If you did an "inst var refs" on a variable of such a class, you'd find two users, one to return the value of the variable and one to set the value.

Points make a good example of why inheritance demands consistent use of accessing methods. Suppose you want to make a subclass of Point that obeyed the same protocols, but stored its location in polar coordinates, as r and theta. You can make such a subclass, but you will swiftly discover that you have to override most of the messages in the superclass because they make direct use of the variables x and y. This defeats the purpose of inheritance. In addition, you would have to be prepared to either declare new variables, r and theta, and waste the space for x and y in your subclass, or store r in x and theta in y and keep track of which is which. Neither is an attractive prospect.

If Point had been written with accessing methods, at least the problem with inheritance would not arise. In your subclass, you could override the messages accessing and setting x and y, replacing them with computations converting polar to Cartesian coordinates and vice versa. At the cost of four methods you would have a fully functioning PolarPoint. A more fully factored solution, one that solves the problem of wasted or misnamed storage, would be to have an abstract Point class with no variables, and subclasses CartesianPoint and PolarPoint.

ACCESSORS—*NOT!*

Many in the Smalltalk community were compelled by this argument (or arrived at the same conclusion independently). Vocal and influential organizations such as Knowledge Systems Corporation made consistent use of accessors a fundamental part of their Smalltalk teaching. Why are there still heathens who refuse to bow to this superior wisdom?

Most easily dismissed is the issue of productivity. All those accessors take too long to write. Most extended Smalltalk environments include support for automatically generating accessing and setting methods. Some are activated when the class is recompiled, asking whether you want accessors for the new methods, others appear when a "message not understood" error occurs, by noticing that the receiver has an instance variable of the same name as the offending message. In any case, writing accessors need not be time consuming.

A slightly more serious argument is performance. All those accessors take time to execute. While it is true that accessing a variable directly is faster than sending a message, the difference is not as great as you might think. Digitalk and ParcPlace are careful to make sure that looking up a method is fast, particularly in common cases like sending a message to the same class or receiver as you did the last time you were in this method. In addition, the CompiledMethod representing the accessor has special flags set to allow it to be executed quickly, without even the overhead of pushing a frame on the stack. In tight loops where the performance of accessors might still be a problem, you can probably cache the value in a temporary variable, anyway.

The crux of the objection is that accessors violate encapsulation. Accessors make details of your storage strategy visible to

continued on page 22...

form portability and second to supporting true native look and feel. Here are the comments I received on performance:

- It would certainly be nice if it ran faster, but I think resources might be better devoted elsewhere. [Speed] might help attract potential new customers, though.
- Speed is very important (that simple).
- Speed is the standard problem with Smalltalk.
- My first major program in Smalltalk (a simulation) still doesn't run fast enough to be useful. Definitely give me more speed.
- Faster execution speed will have a large effect on the use of Smalltalk in industry. Although Smalltalk would be fast enough for their applications, C is often used instead "just in case."
- Execution speed will always be important and [it will] never be [fast] enough, so it needs constant attention.
- The biggest negative perception Smalltalk has from the general computing community is that it is too slow. Unless this perception is corrected, Smalltalk will remain a "cult language." My particular project is a large-scale Smalltalk effort, and I am anticipating execution speed to be a major problem.
- We do some heavy computation using it.

I heard a contrary opinion recently. At the February meeting of the North Texas Society for Object Technology, a speaker from Texas Instruments described a chip fabrication software system they developed using ParcPlace Smalltalk, Gemstone, The Analyst, Envy, and other third-party products. This is a big system with over 3,000 classes. The speaker said that in no case did they encounter a performance bottleneck that was Smalltalk's fault. The problems they did have were due to misapplying the technology. So there are some major users who do not consider performance to be a problem.

COMPARING PARCPLACE SMALLTALK TO DIGITALK SMALLTALK: FIRST TRY

There is considerable data in the literature measuring Smalltalk-80's performance. The green book mentioned previously covers early, experimental implementations. ParcPlace's newsletter publishes Dorado benchmarks for current commercial versions. And the Self group has compared ST80 2.4 to Self 91 and to C.

I haven't seen literature comparing the performance of ParcPlace's Smalltalk-80 with Digitalk's Smalltalk/V. The current article is a modest, if flawed, step in this direction.

People often claim that ST80 is faster than ST/V. Is this true? Recent articles in comp.lang.smalltalk bring this into question. Someone published two very simple benchmarks for the OS/2 versions of ST/V, and others published results for ST/V-Windows and ST80-Windows. The results were surprising, because the 32-bit version of ST/V for OS/2 was, at first glance, between 1.5 and 3 times faster than the 32-bit version of ST80 for Windows. The 16-bit versions of ST/V fared much worse, probably because both benchmarks generated numbers that would be LargePositiveIntegers for 16-bit Smalltalk. Later, I discovered that by slightly modifying one of the benchmarks, ParcPlace moves from being 3 times slower to one third faster than the fastest Digitalk version. It remains 1.5 times slower in the other benchmark. The code as posted follows, and the results are given in Table 2:

1. while loop (original posting)

```
| anIndex |
Time millisecondsToRun: [
    anIndex := 100000.
        [anIndex 0] whileTrue:[ anIndex := anIndex - 1]]
```

2. while loop (modified for ST80 by declaring anIndex as a temporary block variable)

```
Time millisecondsToRun: [
    | anIndex |
    anIndex := 100000.
        [anIndex 0] whileTrue:[ anIndex := anIndex - 1]]
```

3. Fibonacci number generator (tested with "30 fib")

```
fib (in class integer)

    self 1
        ifTrue:[ ^((self - 1) fib + (self - 2) fib ) ]
        ifFalse:[ ^1]
```

Hardware 486/33 (Feldtmann, Samuelson 16MB; Nouwen 8MB).

COMPARING PARCPLACE SMALLTALK TO DIGITALK SMALLTALK: SECOND TRY

As a ParcPlace customer, I was intrigued and startled enough by these results that I decided to measure how the Digitalk and ParcPlace products perform on a wider range of tests. The goals of the benchmarks I developed are:

- Portability between versions of ST80 and ST/V, including ST/V-DOS.
- Writing in as idiomatic a style as portability would allow.
- Being able to compile and run within ST/V-DOS's 640K limit.
- Keeping integers small enough to not skew the results against 16-bit versions.
- Running for a long enough time to get fairly accurate results.
- Being cpu intensive while avoiding accesses to disk or video subsystems.
- Avoiding disk paging.
- Measuring both low-level and medium-level operations.

These goals were to some extent mutually exclusive. For example, it is hard to keep integers and loop counts within the bounds of 16-bit integers while still consuming measurable amounts of time. And it is hard to consume enough time without exceeding runtime resource limits of ST/V-DOS. It took experimentation and dozens of reboots of my machine during ST/V-DOS runs before I arrived at something that met all the goals. The resulting benchmarks are called *slopstones* (Smalltalk Low-level OPeration

...continued from page 1

Table 1. Language execution speed as percentage of optimized C.

Language	Stanford Suite	Puzzle	Richards
ST80 2.4	10%	4.4%	9.4%
Self 91	57%	27%	35%

ference proceedings.[2] They compared the execution speeds on a Sun SPARC workstation of C, Smalltalk-80 2.4, and Self 91 (and some other languages) on the Stanford Suite of integer benchmarks, the Puzzle benchmark, and the Richards operating system simulation benchmark.

The results, given in Table 1, are impressive, especially since Self is at least as hard to optimize as Smalltalk, and the same techniques used to tune it can be applied to Smalltalk. Self actually did better than the numbers indicate. Relative to Smalltalk-80, its optimization doesn't freeze the definition of low-level looping constructs. And it supports several features not found in optimized C: "generic arithmetic, robust error-checking primitives, and support for source-level debugging." In demonstrating an object-oriented environment that is both efficient and full-featured, the authors claim that "programmers no longer need to choose between semantics and performance."

Smalltalk did better on the Richards benchmark in an independent test posted to comp.lang.smalltalk in mid-1991. While the Self group measured Smalltalk-80 2.4 to run at 9.4% of optimized C++ on a Sun 4/260 running UNIX, the poster measured Smalltalk-80 4.0 to run at 27% relative to Borland C++ 2.0 on a 386SX running DOS/Windows. He suggested that the difference could be due to using ST80 version 4.0 rather than 2.4.

Just how practical would it be for the commercial Smalltalk vendors to get its speed up to that of Self? Self's optimizations exact several penalties. First, whereas Smalltalk compiles individual methods incrementally at an almost instantaneous speed, the optimizations performed by Self's compiler slow it down to a compilation speed comparable to C. Second, compilation of "uncommon cases" is deferred, but when it does happen during runtime, it may be somewhat intrusive. Third, Self's code takes between one-third to four times more space than the C code generated for the benchmarks. I haven't seen data on Smalltalk code density, but I would expect it to take less space than C. I'm referring to incremental code density, not to the initial size of the class library. In Self's favor, it might look better when compared to heavily object-oriented programs written in C++ than to procedural programs written in C because of the space C++ uses for virtual function dispatch tables. Fourth, the Self environment requires more space than Smalltalk. The people I've talked with at ParcPlace have the impression that on a 32MB machine, Self pages unacceptably, and that you need at least 64MB to run it comfortably. When I raised this point with a Self researcher, he made two rebuttals. First, Self has not been optimized for space. He cited examples of major savings that could be made. Second, he said that Self runs fine on a 32MB machine and does not require 64MB. He did concede that it still takes more space than Smalltalk, but this is at least partly because it is a research language and the focus of the research has not been to minimize memory requirements.

I have talked with several representatives of both ParcPlace and the Self team in the last couple of years about these issues. My impression is that they're not communicating enough. I think that some of ParcPlace's misgivings about Self could be confirmed or denied by talking more with the Self people. I don't know to what extent Digitalk is in touch with Self. If the commercial vendors decide to focus on performance tuning as hard as the researchers, the communication channels will no doubt open wider!

We'll return later to the question of whether Self's optimizations can be applied to Smalltalk.

Table 2. Looping benchmark results.

Tester	Vendor	Version	Word length, bits*	while loop, milliseconds (avg result) original	modified	fibonacci, milliseconds (avg result)
Nouwen	Digitalk	ST/V-Win 2.0	16	3,570	n/a	32,960
Feldtmann	Digitalk	ST/V-PM 1.4	16	2,530	n/a	9,503
Nouwen	Digitalk	ST/V-PM 1.4	16	2,530	n/a	9,470
Feldtmann	Digitalk	ST/V-PM 2.0	32	125	n/a	4,673
Nouwen	Digitalk	ST/V-PM 2.0	32	120	n/a	4,650
Samuelson	ParcPlace	VW-Win 1.0†	32	353	92‡	6909

* *These are my assumptions about the underlying word length in the virtual machines. Although VW-Win runs on a 16-bit operating system (Win 3.1), it is a 32-bit implementation because it is compiled with a 32-bit DOS extender.*

† *This is ParcPlace's new VisualWorks product, which is ST80 with an interface builder and other extras bundled.*

‡ *Notice the dramatic speed-up for ParcPlace when the declaration of the temporary index variable is moved inside the outer block. ParcPlace distinguishes between clean blocks, copying blocks, and full blocks. These vary, respectively, from fastest to slowest and from least context overhead to most overhead. In moving the variable declaration, the outer block goes from a full block to a clean block and the loop runs four times faster. This confirms ParcPlace's admonition to use clean blocks whenever possible. I don't think Digitalk makes these distinctions, at least for its DOS version.*

HOW IMPORTANT WOULD A FAST SMALLTALK BE TO USERS?

The jury is out on whether Self's optimizations will find their way into Smalltalk and other commercial languages. In the meantime, it would help your vendor to know how much priority you give to performance. I took a survey of ParcPlace customers in the comp.lang.smalltalk newsgroup in October 1991 and asked them to rank the importance of 19 features. Faster execution speed came in third place, with first place going to maintaining cross-plat-

GUIs

Greg Hendley & Eric Smith

Using MS Help from within VisualWorks

The host "looks" that can be achieved with ParcPlace's VisualWorks can be impressive. However, once impressed, a client may ask for even more host–user interface integration. These requests can extend past the look that ParcPlace provides. The client may ask for the feel of the host system. In the case of the Microsoft Windows platforms this may include the ability to run any application without a mouse. This is an anathema for most Smalltalk programmers. Another request under windows might be integration with the help system.

Think about it. While it is not a "widget," the help system is very much part of the user interface. It is the users' way of obtaining more information on how to use an application. The rest of this column will show you how to get started integrating VisualWorks with the help system under Microsoft Windows.

Accessing Microsoft Help from VisualWorks requires knowledge of ParcPlace's Objectkit\Smalltalk C Programming (otherwise known as C Programming Object Kit or CPOK) and Microsoft Help (MS Help), each of which deserves its own column (at least). In this column, we'll explain only enough of each to get you going. The goal is for you to be able to activate MS Help from within VisualWorks and have the help document open on the topic you specify.

MS HELP

The MS Help application (`MSHELP.EXE`) lets you read hypertext-like help files. Help files may contain multiple topics. A *topic* is the unit of information that may be presented at one time by the MS Help application. In your application a topic may provide information on a visual part, a menu, or a window. The Microsoft Help Compiler generates help files from word processing documents saved in Rich Text Format (RTF). Refer to the Microsoft Windows Software Development Kit for more information on generating help files and defining topics.

In your Smalltalk application, you will invoke the MS Help application. Your application can simply activate MS Help, or it can specify the help file and topic that MS Help should open on. MS Help is invoked through the MS Windows API (Application Programming Interface) `WinHelp()`.

CPOK

ParcPlace's Objectkit\Smalltalk C Programming lets Smalltalk access programs written in C. This includes the Microsoft Windows API functions. We will use it to invoke `WinHelp()`. CPOK is a definite improvement over writing your own primitives.

Access to C API functions is through subclasses of ExternalInterface. In general you will create a class for each API and a method for each function. The subclass creation method for ExternalInterface is different from that of most classes.

```
subclass: t
includeFiles: if
includeDirectories: id
libraryFiles: lf
libraryDirectories: ld
generateMethods: gm
beVirtual: bv
instanceVariableNames: f
classVariableNames: d
poolDictionaries: pd
category: cat
```

This method, in addition to creating a subclass, parses header files and creates methods corresponding to the functions defined in the header file. The method also creates methods corresponding to other externals of the header file.

Once ExternalInterface creates the subclass and methods, all you have to do is use them.

USING CPOK TO ACCESS MS HELP

First, we will define the class. Then we will go over how to use it.

Class definition

Create the class `WindowsLibraryInterface` as a subclass of `ExternalLibrarySupport`. If all your files are on your C: drive and your directory structure is similar to ours, your class definition will look something like this:

```
subclass: #WindowsLibraryInterface
includeFiles: '\windows.h'
includeDirectories: 'c:\windev\include'
libraryFiles: 'gdi.exe krn/386.exe user.exe'
libraryDirectories: 'c:\windev\debug'
generateMethods: '*'
beVirtual: false
instanceVariableNames: ''
classVariableNames: ''
poolDictionaries: ''
category: 'ExternalLibrarySupport'
```

An explanation of each of the parameters can be found in the Objectkit\Smalltalk C Programming User's Guide. One parameter is worth explaining here, though. The argument `gm` (`'*'` in the above code) indicates that methods should be generated for all externals, functions, and otherwise. You could instead list just

continued on page 15...

THE BEST OF comp.lang.smalltalk

Alan Knight

Sets and dictionaries

Sets and dictionaries are widely used classes implementing well-known data types. In many ways they are exemplary, as the basic public interface is simple to use, efficient, and corresponds well to the standard abstract data types of the same name. Unfortunately, both classes can present a number of subtle difficulties. Many of these difficulties relate to the fact that both are implemented by hash tables, and that this implementation shows through more than it should.

A good abstract data type is specified without reference to its implementation, and ideally should have several possible implementations, differing only in performance characteristics. The specification should not be written to favour or depend on a particular implementation.

These goals are not always easy to live up to, and Sets and dictionaries fall short in a number of areas.

HASHING

The hashing mechanism provides an efficient search mechanism with little space overhead. It does, however, require the user to provide certain operations. These discussions refer to both dictionary keys and set elements. To save repetition, I'll refer to both as *keys*, and to both sets and dictionaries as *hash tables*.

Any hash table key must provide two methods: = and hash. A simple description of the hashing process follows. For a particular key, the hash method is used to compute an offset into the table. If that slot at that offset contains nil, the key is not present. If the slot is occupied, we test for equality with the key. If the two are equal the search has succeeded. If the two are not equal, the offset will be repeatedly incremented until an object equal to the key or a nil slot is found.

This implementation has a few implications. First, since nil is used to mark empty slots, it cannot be used as a dictionary key or inserted into a set.

Second, objects must provide = and hash methods. More importantly, they must provide these methods such that equal objects have the same hash value. Note that the converse need not hold: Objects with the same hash value do not have to be equal.

The default implementation of = is the object identity test ==, and the default hash method is compatible with this. A common mistake for Smalltalk novices is to define a different equality relation without defining a corresponding hash method. Although this is a well-known mistake, there are similar, more subtle problems.

CHANGING HASH VALUES

A hash function that is not based on object identity will probably be based on instance variables of the object. A common strategy is to add or XOR together the hash values of the significant instance variables, possibly with some additional scrambling. For example, in V/Windows:

```
Point hash
    ^x hash + y hash.
```

ParcPlace Smalltalk has

```
Point hash
    ^(x hash bitShift: 2) bitXor: y hash
```

The problem arises if any of those instance variables are changed. The hash value is then changed, and the object will hash to a different place in a set or dictionary. Any hash tables with that object as a key need to be rehashed, and there is no standard way of finding which tables those are.

This can be a very serious problem and difficult to track down. In practice, however, it doesn't seem to arise all that often. I suspect the explanation lies in the normal usage patterns. The most common dictionary keys are strings and symbols, which are not normally modified. Sets often use a greater variety of objects, but mostly use the default identity-based hash function.

IDENTITY HASHING WITH become:

Even identity-based hashes aren't completely safe, since the become: operation can change them. I've encountered an example of this with a simple version control system in V/Windows. In order to keep track of which added classes belonged to an application, the system maintained a set of classes. Classes do not override = or hash, so they inherit the identity-based version.

In Smalltalk/V Windows, there is a special class DeletedClass. When a class is deleted, the last thing the system does is:

```
classToBeDeleted become: DeletedClass.
```

This achieves two goals. It ensures that classToBeDeleted can be garbage collected, since any references to it have been re-

generated ViewManager subclass. You can add your own events and include them in the list of supported events.

CompositePanes can be nested within one another to any level. If you define tabbing order within your CompositePane, this nests properly as well. However, you must be careful to avoid potentially recursive definitions. WindowBuilder Pro was only able to detect single level recursion (e.g., you can't place a copy of a CompositePane within itself) but it cannot check for later recursion. If you defined A to contain B and vice versa you would be in big trouble. CompositePanes may have one of three styles: default, borders, and scroll bars. The last style is the most interesting. Placing scroll bars on a CompositePane allows you to place widgets within scrolling panes for the first time. While we wouldn't necessarily recommend doing this from a GUI point of view, its nice to know that we *can* do it.

WindowBuilder Pro provides several additional features that simplify working with CompositePanes. If you double-click on a CompositePane, it will open another copy of WindowBuilder Pro on the CompositePane definition itself. If you change the definition, it will change the CompositePane everywhere you have used it. If you decide that you don't want the CompositePane and would rather use its components directly, use the Ungroup command to split them apart losing any "composite" behavior.

OPEN ARCHITECTURE

In addition to adding lots of features for the end-user developer, OSI has also opened the WindowBuilder architecture to make it easier for third parties to build tools that integrate with the product. A new Add-in Manager allows other products to bind themselves to WindowBuilder Pro and add their own functionality and menus. Adding new widgets to the tool palettes is also easy. You must still define support for your widget the same way you would under WindowBuilder. Once you've done that, you create a tool palette bitmap for it and a simple add-in that adds your widget to the Add menu.

PLATFORMS

OSI plans to include a number of follow-on products that integrate with WindowBuilder Pro. They have already announced ENVY/Developer and TEAM/V versions of the product and they plan on having a Macintosh version that is compatible with the current Windows and OS/2 versions.

CONCLUSION

WindowBuilder has been the tool of choice for many Smalltalk/V developers for years. WindowBuilder Pro represents a logical and necessary evolution of the product that should serve the Smalltalk community well into the future. It provides significant new capabilities with its CompositePane technology and adds novel GUI building features such as the Scrapbook and Morphing utilities that should make for a pleasant GUI development environment. ▣

Eric Clayberg is Director of the Computer-Human Interaction Lab at American Management Systems. He is an expert in applying O-O and Smalltalk technology to the design and construction of advanced graphical user interfaces. He can be reached on Compuserve at 72254,2515. S. Sridhar is an independent Smalltalk developer whose interests include building professional quality class libraries. He is affiliated with classAct Technology in Cary, NC. He can be reached on Compuserve at 71031,3240.

■ GUIs

...continued from page 9

those externals essential for bringing up help. Unfortunately, there are dependencies in the externals defined in the windows header file. For a first pass it is easier to use '*' and create all possible methods. Warning: this may take 15 to 20 minutes.

Initiate help on a specific topic

MS Help may be opened on a help file in a number of different ways. To get you started, we will show you how to open on a particular topic. In a workspace, do:

```
WindowsInterface new
WinHelp: self GetActiveWindow
with: 'c:\my-help.hlp'
with: 20
```

where my-help.hlp is your help file and 20 is the context number for a topic in your help file. MS Help will then open on your help file and show the information for topic 20. Keeping track of which topic is which is an interesting issue that you will have to work out for yourself.

The above test code uses the method GetActiveWindow. This method answers the handle of the active window. The method was generated when you created your subclass of ExternalInterface. This is one of the side benefits of using '*' and having all methods created instead of specifying only those that look like they are necessary for help.

CLOSING

We have shown you the essential low-level Smalltalk necessary to get MS Help working with VisualWorks applications. Now you are ready to tackle the higher-level tasks of associating help topics with your windows, menus, and other visual components.

Acknowledgments

We would like to thank our coworkers Kyle Brown, who made using Objectkit\Smalltalk C Programming much easier, and John Cribbs, who applied it to accessing MS Help from Smalltalk. ▣

Greg Hendley is a member of the technical staff at Knowledge Systems Corporation. His OOP experience is in various dialects of Smalltalk. Other experience includes flight simulator out-the-window visual systems. Eric Smith is a member of the technical staff at Knowledge Systems Corporation. His specialty is custom graphical user interfaces using Smalltalk (various dialects) and C. They can be contacted at Knowledge Systems Corporation, 114 MacKenan Drive, Cary, North Carolina 27511, or by phone at 919.481.4000.

on top of their parents (great for floating toolbars), minimize with them, and close when their parents close (very much like MDI without the clipping). Sibling links create a child window of the current window's parent (e.g., your desktop window).

ActionButtons allow you to attach predefined code snippets to a button. Some of these, like "Cancel" come standard with the product. ("Cancel" performs "window close" on any window it sits on). The ActionButton attribute editor lets you select these predefined actions or create your own in standard Smalltalk. Almost any action that is not window specific could be coded once and then reused. WindowBuilder Pro needs to provide a rich variety of these predefined code snippets. The user can modify them appropriately, thus adding to the catalog of these reusable code snippets.

The LinkMenus and ActionMenus function the same way as the LinkButtons and ActionButtons. Any menu option defined with the menu editor may have a link or action associated with it. For example, you can assign the action "Cancel" to the "Exit" menu item.

WIDGET MORPHING

This is a nifty feature that will alleviate the frustration of many a WindowBuilder user. What is widget morphing? It is a feature that allows you to transform a widget from one type into any other while mapping over any common attributes.

To demonstrate how useful this is, suppose you create a ListBox, give it a name, attach a list, set its color and fonts, and give it a few event handlers. Later, let's suppose you discover that your window doesn't have room for a ListBox and you opt to use a ComboBox instead. Before the advent of this feature, you would have had to add a new control and copy all of the original control's attributes to the new control by hand (or you could change the WindowBuilder generated code by hand, which is *verboten*). Now, you can accomplish the same thing by clicking on the widget with the right mouse button and selecting the "Morph" option. This presents a cascaded list of all "similar" widget types (e.g., ComboBoxes, ListPanes and MultiSelectListBoxes in the case of ListBoxes) as well as an "Other . . ." choice (for those rare occasions when you want to transform a ListBox into a totally different widget, such as a button). Choose the one you want and your widget transforms instantly. Only events that both the old and new widget understand will be mapped over; the new widget will acquire as many of the original widget's attributes as it understands and default the rest. Be careful when morphing widgets, because while all widgets respond to #getContents, they expect very different things. It would be nice if WindowBuilder Pro added a warning message when potentially troublesome morphing is attempted.

SCRAPBOOK

One of our favorite new features is the Scrapbook. Anyone who has used the Macintosh will appreciate this one right away. (Actually anyone who has ever had to reuse visual components will appreciate this right away). The Scrapbook provides a place to store fully defined widgets or sets of widgets. It allows you to organize your creations in multiple chapters containing multiple pages. Each page contains a user-defined object.

Start by creating and defining a group of widgets. Select them all and select the Store option from the Scrapbook menu. Name your creation and select one or more chapters in which to place it. You can organize your objects under as many categories as you like. New chapters can be created with the touch of a button. There is a single special chapter entitled "Quick Reference." Anything added here is automatically appended to the "Scrapbook . . . Quick Reference" cascading menu for instant access.

In order to retrieve something from the Scrapbook, select "Retrieve." You are then presented with a listing of all of your chapters and pages. Clicking on any page will display its contents in a graphic view to the right. This allows you to preview any object before placing it on the screen. Selecting a page and hitting OK loads the cursor with the selected object which you can then drop anywhere you like.

You can easily save Scrapbooks to disk and retrieve them. Each developer can have a Scrapbook, and these can then be merged together to provide a common set of components across a development team.

CompositePanes

While the Scrapbook provides a repository for storing reusable visual components, WindowBuilder Pro's new CompositePane technology provides the mechanisms to actually create these reusable visual components. In Smalltalk, we routinely build complex classes by synthesizing structure and behavior from simpler classes. In a like manner, CompositePanes allow you to create compound or composite widgets out of other atomic widgets. WindowBuilder Pro includes an example of this in a sample CompositePane subclass called SexPane. A SexPane is composed of three widgets: two RadioButtons (Male and Female) and a GroupBox (labeled Sex). WindowBuilder Pro treats it like any other standalone widget. If you resize it, its components resize relative to itself. It even has its own instance variables and events. For example, in response to a #sexChanged event (issued whenever the user clicks one of the RadioButtons), you could bring up a MessageBox announcing the new state. Setting its contents is as simple as sending the message: aSexPane contents: #male.

OSI has seamlessly integrated this functionality with the rest of the product.

To create a CompositePane, select the appropriate option from the File menu or select several existing widgets that exist in your editing window and select the Create Composite command. This opens a new copy of WindowBuilder Pro with the selected components in it. (Here the WindowBuilder Pro itself acts as an attribute editor for the CompositePanes. Neat!). Give them names or further define them anyway you like. When you save them you are prompted for a class name and a superclass (generally CompositePane). WindowBuilder Pro creates the class and then enquires whether you would like to replace the original widgets with the new composite. Once you have a CompositePane subclass defined you may add code to it exactly the same way you would add code to a WindowBuilder

moved. It also ensures that any code which referenced classToBeDeleted will report an error when executed.

Unfortunately, if a class is removed outside the framework of this version control system, any applications that contained it now contain references to DeletedClass. Further, those references are stored according to the hash value of classToBeDeleted, so they can't be removed using the public set interface.

In order to remove DeletedClass, the set must be rehashed.

As a final difficulty, V/Windows does not provide a rehash operation. Fortunately, for this application, the slow-and-dirty implementation

```
aSet become: aSet copy
```

is sufficient.

HASHING PERFORMANCE

Even if your hash function doesn't play tricks on you, defining one with a good distribution can be difficult. Jeff McAffer (jeff@is.s.u-tokyo.jp) writes:

> **I was recently looking at a system that made extensive use of sets . . . . One of the benches was putting a whole bunch of two-element arrays into the sets. It turns out that 60% of the processing time was in the set hashing. The cause? In V/Win (likely all Vs) the hash function for arrays returns the receiver's size. I changed the hash function and doubled the speed of the benchmark.**

The identity-based hash function usually has a good distribution, but has a relatively small number of significant bits.

> **"Performance will suffer greatly any time a hash table contains more elements than the hash function can handle well."**

Bruce Samuelson (bruce@ling.uta.edu) writes:

> **I think the IdentityDictionary hash function runs out of steam at about 14 bits (16K objects).**

Performance will suffer greatly any time a hash table contains more elements than the hash function can handle well. To help determine if this is the case, Bruce Samuelson has also written a method to measure dictionary hash performance. It is written for ParcPlace Smalltalk, but should be easily adaptable to Digitalk dialects, and is available from either the Manchester or Illinois Smalltalk archives, under the title *dictionary-performance:*

```
!Dictionary methodsFor: 'statistics'!
hashStatistics
"This method tests how well the receiver is hashed.
It is adapted from
<Dictionary findKey0rNil:>."
"Smalltalk hashStatistics"
    "Return an array:
    at: 1    basicSize of dictionary
    at: 2    size of dictionary, i.e., number of elements (associations)
    at: 3    average miss of hash function
        0 means hash is ideal
        N means avg element is placed N steps beyond its hash value
        large number means hash is bad
    at: 4    histogram (using a sorted collection) of misses"

    | basicSize size total histogram |
    basicSize := self basicSize.
    size := self size.
    total := 0.
    histogram := Bag new.
    self keysDo: [:key |
        | miss location probe |
        miss := 0.
        location := key hash \\ basicSize + 1.
        [(probe := self basicAt: location) isNil or: [probe key ~= key]]
            whileTrue: [
```

```
            miss := miss + 1.
            (location := location + 1) > basicSize
                ifTrue: [location := 1]].
            histogram add: miss.
            total := total + miss].
^Array
    with: basicSize
    with: size
    with: (total / (size max: 1)) asFloat
    with: histogram sortedElements! !
```

LARGE INSTANCES

There are other factors that might affect the performance of hash tables. For example, very large arrays of pointers (most collections, but not ByteArrays or WordArrays) can cause problems for the garbage collector. Earlier versions of ParcPlace Smalltalk included an arbitrary limit of 100,000 on the size of such collections. They've removed the limit, but the problem remains. The source of the problem is the copying garbage collectors used in Smalltalk, which can be forced to spend a lot of time copying these large objects back and forth.

> "For very small dictionaries, it may not be necessary to use a dictionary at all."

Is poor performance on very large hash tables a problem? It's certainly not the common case. **Rik Fisher Smoody** (riks@ogicse.cse.ogi.edu) writes:

> Consider Dictionaries. The overhead of creating a small one is small. This is good. I checked one handy image: there were 540 instances of dictionary or subclasses with a total of 4,137 elements...an average of less than 10 objects/dictionary.
>
> But occasionally a giant arises.... What if there were a class called BigDictionary that obeys all of the external protocol of Dictionary, but is tuned for performance when it is large? Perhaps when a small (ordinary) dictionary grows, it could automatically turn into a BigDict.

Very large hash table performance is one of those things you don't usually worry about, but when you do need it, it's very important. A BigDict would be a very handy thing to have, and I'm sure there's already more than one implementation out there. **Jan Steinman** (steinman@ascom.hasler.ch) writes:

> To get a start on this, look at the Symbol class variable USTable, which is sort of an ordered BigSet, although it isn't implemented as a class. The general strategy is divide and conquer, as in KSAM.

USTable (I looked at ParcPlace R4.1) seems to be a bucketed hash table with some code for choosing good dictionary and bucket sizes. The buckets are weak arrays, which stops USTable from holding on to otherwise unreferenced symbols. It may also improve speed, since weak arrays have some additional searching primitives.

Divide and conquer normally means splitting a problem up into sub-problems, each of which can be solved more easily than the whole and reassembled to form a solution to the complete problem. For a large set, the obvious decomposition is into smaller sets. By converting USTable's buckets into sets (or IdentitySets), it would be easy to convert this into a divide and conquer solution that would help avoid the performance problems of very large hash tables.

SPACE OVERHEAD

Most dictionaries are small, so the performance problems of large hash tables don't affect them. Applications that use many small dictionaries can, however, suffer from serious space problems. In particular, regular dictionaries are implemented using associations, which requires another object with two instance variables for each element in the dictionary.

IdentityDictionaries are implemented without associations in both ParcPlace and Digitalk versions. ParcPlace uses two parallel arrays of keys and values. Digitalk uses one array, storing keys at odd indices, values in even indices. Both are much more space efficient than normal dictionaries, but make operations that access associations (e.g., associationsDo:) much slower.

I'm not sure why this particular choice was made. It's nice to have more space-efficient dictionaries, but I don't see why that should be coupled to the use of identity versus equality.

For very small dictionaries, it may not be necessary to use a dictionary at all. If the number of keys is a small constant, a class using linear search may be just as efficient in time, and save even more space (this would have much less impact than regular vs. identity dictionaries). Lazy initialization can help enormously if not all objects have properties.

CONCLUSION

I've shown a few examples of problems that can arise using the hash table classes in Smalltalk. There are other tricks, such as assuming the identity of associations in a dictionary remains constant and retaining or modifying them. I think this is a bad thing, but the base Smalltalk system does it, so it's not likely to disappear soon. A broader issue is that some people believe the association-based nature of dictionaries is too public and that this imposes excessive costs on other implementations (such as IdentityDictionaries). A future column may explore these and other issues. ■

Alan Knight works for The Object People. He can be reached at 613.225.8812, or by e-mail as knight@mrco.carleton.ca.

SNEAK PREVIEW

Eric Clayberg & S. Sridhar

WindowBuilder Pro: new horizons

GUI builders have become *de riguer* in the PC desktop computing marketplace. For the past few years, WindowBuilder from Cooper and Peters has been the primary tool for building Smalltalk/V-based GUI applications. At the beginning of 1993, C&P decided to get out of the Smalltalk market. A new company, Objectshare Systems Inc. (OSI), took over the responsibility of marketing C&P's WindowBuilder line of products. WindowBuilder is the premier tool for Smalltalk/V GUI development. WindowBuilder is designed to coexist with the standard Smalltalk/V environment and, as such, generates human-readable class definitions and message interfaces. To meet the ever-increasing demands of sophisticated GUI applications, OSI is evolving the WindowBuilder product line into a professional version of the GUI builder called the WindowBuilder Pro.

As early beta testers, we'll report in this article on a number of the new features and enhancements that are an integral part of WindowBuilder Pro. Because WindowBuilder was reviewed in one of the very first issues of THE SMALLTALK REPORT, we'll skip over all its basic features.

NEW LOOK AND FEEL

WindowBuilder Pro has a nicer look and feel than WindowBuilder. Colorful toolbars abound. Across the top of the screen are buttons for creating new windows; theses include Cut, Copy, Paste, Alignment, Distribution, and Z-Order Control among others. A duplicate command that works like the corresponding command in MacDraw is a new feature. Selecting a widget or collection of widgets and hitting Duplicate creates a copy offset from the original. Moving the copy relative to the original and hitting Duplicate again results in more copies at the new offset.

WindowBuilder Pro provides increased access to the Font, Color, Framing, Menu, and other commands. Although the commands work the same way the did before, they are now accessible through a toolbar and via pop-up menus. The toolbar is right below the main editing area, and you can access the pop-up menu by clicking the right mouse button over any widget. The Framing editor has been slightly enhanced to allow users to lock objects to the horizontal and vertical centerlines of a window (as opposed to just the right, left, top, or bottom sides).

Next to the attribute toolbar are two new items that VisualBasic fans will appreciate: size and position indicators. As you move or resize widgets, these indicators constantly update to reflect the new information. This feature is very useful for precise work. A new status line at the bottom of the screen gives context-sensitive help. As you drag through menu commands or over toolbar choices, the status line describes each option. As you drag through the widget tool palettes, it describes each widget type. This is especially helpful for those whom the "intuitive" meaning of the many icons is not so intuitive. Also, as you click on any object in the editing window, the status line identifies its name and type.

In addition to the new look, WindowBuilder Pro has several nice ergonomic enhancements. You can now leave autosizing on all the time. StaticText, Buttons, CheckBoxes, and RadioButtons will automatically autosize as you type in labels. StaticText autosizes in the proper direction depending on its style. (The right-justified labels now autosize to the left! This should eliminate many of those type-autosize-move sequences). Autosizing now also conforms to the grid, rectifying an annoying oversight in the original WindowBuilder.

All widgets now include an attribute editor, and all widgets draw correctly in the edit pane (no more generic rectangles). ListBoxes and ComboBoxes feature a list editor that allows users to enter an initial list of items. Although this is not useful in cases where dynamic list data is needed, it is handy during rapid prototyping or when the items are static. DrawnButtons and StaticGraphics can now display a bitmap in the editing window. In the field for entering text for a widget, you enter the name of the bitmap file (.BMP) that you would like to use. If WindowBuilder Pro finds the file, it will display it for you. Your other option is to double-click to bring up a file dialog from which to select a bitmap. The application window's attribute editor also has been enhanced to allow the addition of minimization icons to window definitions.

RAPID PROTOTYPING

WindowBuilder Pro has four new components to facilitate rapid prototyping. These are the LinkButton, ActionButton, LinkMenu, and ActionMenu. These components provide easy ways to link windows together and perform simple actions without writing any code. LinkButtons provide a way to hook windows together without writing any code. Place a LinkButton on the screen and double-click on it to see a list of all of your ViewManager subclasses. Pick the subclass you want, then select the type of link you want. There are three types of links, *independent, child,* or *sibling.* Independent links have no logical dependency on the window that created them. Child links create windows that float

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS MAIL PERMIT NO. 279 DENVILLE NJ

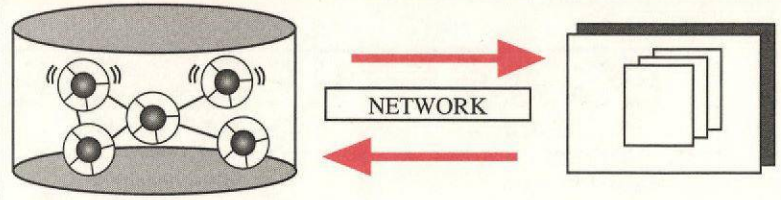
POSTAGE WILL BE PAID BY ADDRESSEE

The Smalltalk Report
Subscriber Services Dept SML
PO Box 3000
Denville NJ 07834-9821

If You Use Smalltalk, You Need GemStone.

GemStone is the ideal database environment for supporting Smalltalk applications. It is the only high-performance, production-ready ODBMS with a transparent Smalltalk interface.

- Maintain class hierarchies and execute Smalltalk methods directly in the server.
- Automatic, transparent translation of Smalltalk objects into GemStone.
- Cooperative client-server support.
- Smalltalk-based DDL/DML.
- High-performance, scalable, production-ready ODBMS.
- Integrated garbage collection of persistent Smalltalk objects.



❑ **YES! Send Me Complete Details On GemStone**

Name: _____ **Title:** _____

Company: _____

Address: _____

City: _____ **State:** _____ **Zip:** _____

Phone: _____ **Fax:** _____

1-800-243-9369

SERVIO

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS MAIL PERMIT NO. 4362 SAN JOSE, CA

POSTAGE WILL BE PAIDBY THE ADDRESSE

SERVIO CORPORATION
2085 HAMILTON AVENUE
SUITE 200
SAN JOSE, CA 95125-9985

The Smalltalk Report

☞ **Provides objective & authoritative coverage on language advances, usage tips, project management advice, A&D techniques, and insightful applications.**

"If you're programming in Smalltalk, you should be reading **The Smalltalk Report"**

❑ **Yes, I would like to subscribe to The Smalltalk Report** Date_____

❑ 1 year (9 issues)
- ❑ Domestic $69.00
- ❑ Foreign $94.00

❑ 2 year (18 issues)
- ❑ Domestic $128.00
- ❑ Foreign $178.00

Method of Payment
- ❑ Check enclosed (payable to **The Smalltalk Report**)
- ❑ Bill me
- ❑ Charge my: ❑ Visa ❑ Mastercard ❑ Amex

Card No. _____
Exp. Date _____
Signature _____

Name _____
Title _____
Company _____
Address _____
City _____
State _____
Zip _____
Country _____
Phone _____

1. Which dialect of Smalltalk do you use:
- ❑ Smalltalk V
- ❑ Smalltalk-80
- ❑ Other _____

2. What is your involvement in software purchases for your department/firm:
- ❑ Recommend Need
- ❑ Specify Product
- ❑ Make Purchase
- ❑ None

3. Which operating system supports your software:
- ❑ UNIX
- ❑ DOS
- ❑ OS/2
- ❑ Windows
- ❑ Other _____

4. What is your company's primary business activity:
- ❑ Computer/Software Development.
- ❑ Manufacturing
- ❑ Financial Services
- ❑ Government/Military/Utility
- ❑ Educational/Consulting
- ❑ Other _____

5. For how long have you been using Smalltalk:
- ❑ Less than one year
- ❑ 1–3 years
- ❑ 3+ years

E3FG

A member of the Object Marketing Network

fax to 212/274-0646

SIGS PUBLICATIONS

The Smalltalk Report

The International Newsletter for Smalltalk Programmers

July-August 1993 Volume 2 Number 9

SMALLTALK DEBUGGING TECHNIQUES

By Roxie Rochat & Juanita Ewing

Contents:

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

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

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

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

The Smalltalk/V expressions have been tested under:

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

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

WHO AM I?

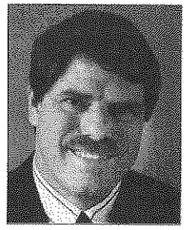
A major component of the debugging process is the collection of information about the objects and their current state. Transcript messages allow you to gather information about objects over time. Inspectors allow you to see objects and their internals in a static state. Careful planning with respect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

hello world, or printf, in Smalltalk

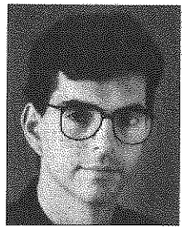
The Smalltalk equivalent of the C printf function is to write to the Transcript window. (You *do* leave your Transcript open, don't you? The system writes error messages to the Transcript so if you collapse or close it, sooner or later, you'll be sorry.)

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

continued on page 18...



John Pugh



Paul White

EDITORS' CORNER

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

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

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

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

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

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

John Pugh Paul White

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. Mailed First Class. Subscription rates 1 year, (9 issues) domestic, $65, Foreign and Canada, $90, Single copy price, $8.00. POSTMASTER: Send address changes and subscription orders to: THE SMALLTALK REPORT, Subscriber Services, Dept. SML, P.O. Box 3000, Denville, NJ 07834. Submit articles to the Editors at 91 Second Avenue, Ottawa, Ontario K1S 2H4, Canada. For service on current subscriptions call 800.783.4903. Printed in the United States.

PRODUCT ANNOUNCEMENTS

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

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

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

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

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

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

Highlights

Excerpts from industry publications

COBOL TO OOP

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

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

THEY SAY WE HAVE A REVOLUTION

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

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

CORBA

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

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

OBJECT SQL

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

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

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

Debugging objects

Bob Hinkle, Vicki Jones, and Ralph E. Johnson

As the premier object-oriented programming language, Smalltalk should give programmers easy access to objects. However, during debugging it can be very difficult to get your hands on a particular object. For example, suppose you're developing a program that stores some objects in an OrderedCollection, but when it tries to retrieve them later, some are missing. You might like to add debugging code to OrderedCollection methods such as add: and remove: to detect when objects are taken out of the OrderedCollection, but any changes would affect every OrderedCollection in the system, bringing your image to a crashing halt. This article will show how to solve this and similar problems by letting you modify code and add breakpoints that affect only one particular object, rather than all objects in a given class.This approach of defining only object-specific methods is similar to what Kent Beck has described.[1,2] Our solution relies on the use of a new kind of class and on some small but powerful variations on CompiledMethods and Compilers. Besides being useful in their own right, we feel these extensions again illustrate (as in our previous articles[3,4]) how powerful Smalltalk's reflective features are, as they allow programmers to adapt and extend the environment to suit their needs. The solution described here is specific to Smalltalk-80, since it relies on Smalltalk-80's architecture for classes, metaclasses, the compiler, and compiled methods and on the complete availability of source code for these system elements.* As a result, our extensions may not apply to Smalltalk Versions, although something similar may be possible.

LIGHTWEIGHT CLASSES

The first step to debugging objects is to be able to modify methods on a per-object basis. In Smalltalk, methods for an object are defined in that object's class and are stored in the class's method dictionary. To change a method for a particular object requires that the object have its own private class. We will give an object that we want to debug its own class by inserting a new class between the object and its real class. We could create a (perhaps temporary or anonymous) instance of class Class for this purpose, but that's a little heavy-handed: Instances of Class have many instance variables and a lot of behavior aren't needed for our purposes. For example, Class adds variables and functionality to define new class and pool variables. In addition, Class inherits from ClassDescription variables and code to support adding new instance variables and class organizations. All of this is unnecessary for a lightweight class, so we defined LightweightClass to be a subclass of Behavior. Behavior is the superclass of ClassDescription, and it defines the code needed for the interpreter to do method lookup. (For more information on the roles of Behavior, Class, and ClassDescription, refer to the chapter titled "Protocol for Classes" in Reference 5.) Since Behavior is a simpler starting point, instances of LightweightClass will be smaller than instances of Class and will require less memory and time to allocate, initialize, and finalize. That makes it easier and less expensive to create lightweight classes on the fly to modify, even if only temporarily, some object's behavior.

Before explaining LightweightClass in detail, it's helpful to review the way things work normally in Smalltalk. When an object is sent a message, the system tries to find a method corresponding to the message's selector in the method dictionary of the object's class. If no such method exists, the system will look in that class's superclass, and so on up the chain of superclasses until a method is found or the end of the chain is reached. Furthermore, when a method is added to a class or changed, the new code is compiled by an instance of the class's compilerClass (which by default in the system is SmalltalkCompiler). The result of compiling is an instance of CompiledMethod, which will be stored in the class's method dictionary with its selector as its key. The source code for the method is not stored directly in the CompiledMethod, but, instead, is written into the change log, and the CompiledMethod is given a pointer to its file and offset.

Our implementation of lightweight classes changes this normal scenario in three ways. The first and most important change inserts a LightweightClass in between an object and its real class (or what we will call *original class,* since it was the class by which the object was originally created), with the object's class being changed to the LightweightClass, and the LightweightClass' superclass set to the object's original class. In this way, any message sent to the object will first be looked up in the LightweightClass's method dictionary. If a method is found there, it will be used to respond to the message, and it will be unique to that particular object. Otherwise, message lookup will continue to the LightweightClass' superclass—the object's original class—and, hence, will proceed as usual for objects of that class. Figure 1 illustrates this relationship between an object, its original class, and its lightweight class.

* Source code for the object debugging package is available by anonymous ftp from st.cs.uiuc.edu. Look for the file ObjectDebugging.st in pub/st80_r41.

method corresponding to aSelector isn't defined in the receiver, a new BreakpointMethod is created and installed in the receiver's method dictionary.

As with lightweight classes, we need a new compiler class, BreakpointCompiler, to implement breakpoints. Once again, though, this class is almost trivial, since it only needs to define newCodeStream to return a CodeStream that creates BreakpointMethods.

PUTTING THINGS TOGETHER

To exploit the functionality provided by LightweightClass and BreakpointMethod, we adapted the interface to make object debugging as simple as possible. This required changing the existing Browsers, adding a menu option to Inspectors, and creating a new Browser specifically for lightweight classes.

The existing Browsers were changed by adding a breakpoint option to the menu in the selector view. Choosing this option will either set a breakpoint on the selected method or, if the method is already breakpointed, remove the breakpoint, so that the option acts like a toggle switch. Furthermore, the selector view allows method selectors to be formatted, and we use a preceding asterisk to quickly distinguish methods with breakpoints.

In addition, all Inspectors now have a new menu option called browseLightweight. Choosing this option will create a new lightweight class for the selected object and open a LightweightClassBrowser to examine and modify methods for that particular object.

LightweightClassBrowser is a subclass of Browser for looking at lightweight classes. As shown in Figure 3, the LightweightClassBrowser has six subviews. The first two views allow you to decide what methods you'll see: You can either see only methods defined in the lightweight class, or all methods up to some specified superclass. The upper right view shows which class you're listing methods up to, while the upper left view shows which class the selected method is actually defined in. This option makes it easy to view a superclass method and then make changes to save in the lightweight class. The third view lists all selectors from the lightweight class up to the class chosen in the upper right view. These selectors are formatted so that all breakpointed methods are marked with an asterisk, and so that all methods actually defined in the lightweight class (as opposed to one of its superclasses) are printed in bold. The fourth view is a TextView on the code of the currently selected method. Finally, the last two views belong to an Inspector on the object whose lightweight class is being browsed.

This interface makes it easy to imagine how the debugging session mentioned in the introduction would proceed. Once you've decided there is a problem with one of your OrderedCollections, you can use a Browser to put a breakpoint on the method where the OrderedCollection is created. When that method is executed, a Debugger will pop up. The Debugger lets you inspect the OrderedCollection and choose the browseLightweight option to create a lightweight class for it. The LightweightClassBrowser lets you put breakpoints on the add: and remove: methods. After you "proceed" from the Debugger, you'll be able to watch as that one OrderedCollection is modified, and you can find out when objects are added to it and when they're removed. With that information, you'll be well on your way to solving the problem.

These changes significantly improve debugging in the Smalltalk environment. Though breakpoints are convenient, it's the functionality of lightweight classes that makes the key difference, as they allow you to monitor or alter the behavior of particular objects without affecting the rest of your system. The changes described here, while not complex, are remarkable in one sense, because they rely on our ability to modify parts of the Smalltalk system that in some languages would be internal and unavailable to programmers. The fact that classes are first-class objects—which is to say, classes are accessible to and modifiable by the programmer—allowed us to introduce a new kind of class and to replace an object's class on the fly during execution. Similarly, we were able to create two subclasses of CompiledMethod, and make an important change to that class itself, only because compiled methods are first class. Finally, Smalltalk's representation of the Compiler itself, and its good design for pluggability, allowed us to create two simple subclasses by defining only one method each. The combination of the ease of making these changes with the significant benefits they provide is a good argument for the desirability of this level of reflection in a programming system. In our next article, we plan to explore one level deeper into Smalltalk's reflectiveness by changing the compiler and the interpreter to introduce active variables and watchpoints. ▣

References

1. Beck, K. Instance-specific behavior, part I, THE SMALLTALK REPORT 2(6), 1992.
2. Beck, K. Instance-specific behavior, part II, THE SMALLTALK REPORT 2(7), 1992.
3. Hinkle, B. and R. E. Johnson. Taking exception to Smalltalk, part 1, THE SMALLTALK REPORT 2(3), 1992.
4. Hinkle, B., and R. E. Johnson. Taking exception to Smalltalk, part 2, THE SMALLTALK REPORT, (2)4, 1993.
5. Goldberg, A., and D. Robson. SMALLTALK-80: THE LANGUAGE AND ITS IMPLEMENTATION, Addison-Wesley, Reading, MA, 1983.
6. A. H. Borning. Classes versus prototypes in object-oriented languages, PROCEEDINGS OF THE ACM/IEEE FALL JOINT COMPUTER CONFERENCE, Dallas, TX, November 1986, pp. 36–40.

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

...continued from page 6

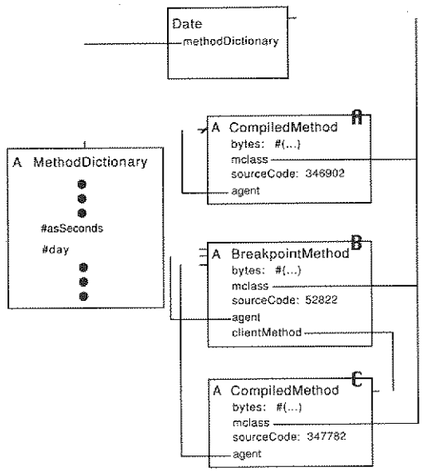


Figure 2. The relationship between CompiledMethods and the BreakpointMethods that represent them.

pointMethod itself is invisible in the debugging process, since it is removed from the execution stack before the debugger opens. In addition, BreakpointMethods implement the getSource message by returning their client's source, and so breakpointed methods can be browsed directly.

The new variable agent is needed to make CompiledMethods with breakpoints print out well. Every CompiledMethod has an instance variable called mclass, which refers to the class in whose method dictionary the CompiledMethod should be found. When CompiledMethods print themselves out, they look in their mclass to make sure they really are defined there ; if they aren't, they will print out as an unboundMethod. Since BreakpointMethods replace their client in the method dictionary, all breakpointed methods would print out as unboundMethods, which is confusing and aesthetically unpleasing. We solved this problem by adding agent:. Now, when a CompiledMethod prints out, it checks to make sure that its agent is defined by its mclass, and if so it prints out normally. Most CompiledMethods are their own agents, but breakpointed methods will have their agent set to the BreakpointMethod that's representing them, and so they'll print out correctly. Figure 2 illustrates this relationship between CompiledMethods and the BreakpointMethods that represent them.

In Figure 2, the asSeconds method for Date—the CompiledMethod marked A—is a normal method. Its mclass is Date, it is its own agent, and it is referred to directly by Date's method dictionary. However, a breakpoint has been placed on the day method for Date. The #day entry in Date's method dictionary refers to the BreakpointMethod B, whose clientMethod is the CompiledMethod C. CompiledMethod C, in turn, refers to BreakpointMethod B as its agent. This way, even though CompiledMethod C is not referenced by Date's method dictionary, its agent—BreakpointMethod B—is, so CompiledMethod C will print as a well-defined method rather than as an unbound one.

We added breakpoints to the system by creating three new methods in Behavior, thus making breakpoints in all kinds of classes, including instances of both Class and LightweightClass. The first method, isBreakpointAt:, tells whether the specified method in the Behavior has a breakpoint set or not. The second, breakpointCompilerClass, returns BreakpointCompiler, which is the compiler used for all classes to create new breakpointed methods. The third method, setBreakpointAt:, is the main one and is used to set or remove a breakpoint. It's implemented as:

```
setBreakpointAt: aSelector
| c m |
c := self whichClassIncludesSelector: aSelector.
c isNil ifTrue: [^self].
m := c compiledMethodAt: aSelector.
self == c
    ifTrue: [
        m isBreakpoint
            ifTrue: [m client mclass == self
                ifTrue: [self addSelector: aSelector
                    withMethod: m client]
                ifFalse: [self removeSelector: aSelector]]
            ifFalse: [self addSelector: aSelector withMethod:
                (BreakpointMethod on: m
                    selector: aSelector
                    inClass: self)]]
    ifFalse: [
        m isBreakpoint ifTrue: [m := m client].
        self addSelector: aSelector withMethod:
            (BreakpointMethod on: m selector: aSelector inClass: self)]
```

If the receiver Behavior is the class that defines the method corresponding to the parameter selector and if the method is already breakpointed, the code removes the breakpoint by testing whether the BreakpointMethod's client is defined in the receiver or not. If it is, the BreakpointMethod is replaced by its client in the receiver's method dictionary; but if it isn't, the BreakpointMethod is simply removed from the receiver's method dictionary (thus leaving the client in whatever other method dictionary it resides). If the method isn't breakpointed, the code creates a new BreakpointMethod for it and adds it to the receiver's method dictionary. Finally, if the

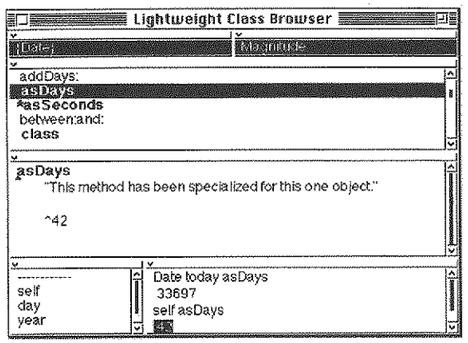


Figure 3. The lightweight class browser.

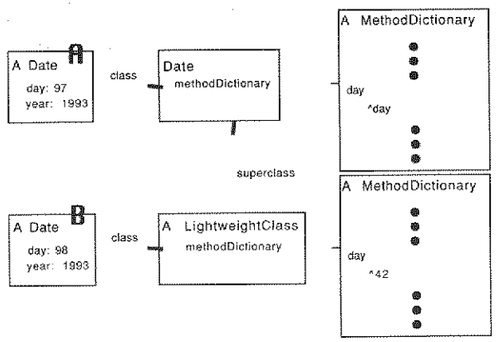


Figure 1. The relationship between an object, its original class, and its lightweight class.

In Figure 1, when the day message is sent to the object marked A, a corresponding method is searched for starting in Date, the object's class. This method returns the value of the day instance variable, which (for A) is 97. However, when the day message is sent to object B, message lookup begins in its class, which is an instance of LightweightClass. The method in the lightweight class's method dictionary is defined to return 42. Thus, object B behaves differently from A and all other instances of Date.

The two other changes pertain to source code management. The code for methods in lightweight classes can't be stored in the change log, since the lightweight class isn't named in the system dictionary, and it has no category or protocols like normal classes. (And in any case, the lightweight class may be an entirely dynamic object that is created while running a program, but which does not persist from one programming session to the next, so that storing code for it in the change log would make no sense.) Instead, we store the code directly with the method it produces, which required us to create a new kind of compiled method, CompiledMethodWithSource. Finally, to produce these kinds of compiled methods, we exploited the "pluggability" of the compiler and created a new subclass of SmalltalkCompiler. We'll describe these two changes after first looking at LightweightClass in detail.

As a subclass of Behavior, LightweightClass adds only one instance variable, name, which is convenient for telling lightweight classes apart. In addition to accessor methods for this variable, LightweightClass defines three other methods of interest: initializeWithSuper:, which initializes a new lightweight class; compile:notifying:ifFail:, which adds a new method to a lightweight class; and compilerClass, which defines the kind of compiler to use for methods in a lightweight class.

A new lightweight class is normally created by sending becomeLightweight to an object. This method is defined in Object as follows:

```
becomeLightweight
| lightweightClass |
self lightweightClass isNil
    ifTrue: [
        lightweightClass :=
            LightweightClass newWithSuper: self class.
        self changeClassToThatOf: lightweightClass basicNew]
```

If the receiver of this message already has a lightweight class, nothing more is done. Otherwise, newWithSuper: is sent to create a new lightweight class whose superclass will be the receiver object's original class. The message changeClassToThatOf: is then sent to the receiver to insert the lightweight class before the object's original class. Because some objects (notably immutable objects like SmallIntegers, Characters, true, and false) can't have their class changed, becomeLightweight can't be sent to them, but it can be sent to all others.

The newWithSuper: method creates a new lightweight class and then sends it the initializeWithSuper: message, where the parameter is the object's original class. This initialization method gives a default name to the lightweight class, creates a new method dictionary for it, and sets its superclass to be the class passed in, so that any messages not found in the lightweight classquote method dictionary will be looked up in the object's original class.

The solution described in the preceding paragraphs makes sure that messages sent to a lightweight object are first looked up in the object's lightweight class as desired. However, class messages will not work correctly as the solution has been presented so far. For example, if aDay is a lightweight instance of Date, sending "aDay class nameOfDay: 1" should be the same as sending "Date nameOfDay: 1," but aDay's class is an instance of LightweightClass, so "aDay class nameOfDay: 1" will try (and fail) to find a method for the message nameOfDay: defined for

LightweightClass. This problem exists because classes have several roles, including roles as method repositories and as repositories for shared information (in this case, the names of the days of the week). We want the lightweight class to play the first role and the object's original class to play the second, but Smalltalk expects one entity to play both roles. (Alan Borning summarizes the various roles of class and suggests an alternative approach in Reference 6.) Our solution to this problem is to separate out the role of method repository, which we did by creating a new method for all objects called dispatchingClass. The definition of dispatchingClass in Object is the same as that of class—it uses a primitive to directly access the object's class from the object's memory structure. When an object is made lightweight, its lightweight class is stored in the memory structure and thus returned as the value of dispatchingClass. In addition, LightweightClass overrides the class method to be:

```
class
    ^self dispatchingClass superClass
```

This will return the object's original class, as desired, since newWithSuper: installed the original class as the lightweight class's superclass.

The LightweightClass method compile:notifying:ifFail: is needed when a method is defined in a lightweight class and is implemented as:

```
compile: code notifying: requestor ifFail: failBlock
    "Compile the argument, code, as source code in the context of the
    receiver and install the result in the receiver's method dictionary.
    The argument requestor is to be notified if an error occurs. The
    argument code is either a string or an object that converts to a string
    or a PositionableStream on an object that converts to a string. This
    method *does* save the source code. Evaluate the failBlock if the
    compilation does not succeed."
| methodNode selector save method oldMethod |
save := code asString copy.
methodNode := self compilerClass new
                        compile: code
                        in: self
                        notifying: requestor
                        ifFail: failBlock.
selector := methodNode selector.
method := methodNode generate.
method sourceCode: save.
oldMethod := self compiledMethodAt: selector ifAbsent: [nil].
(oldMethod notNil and: [oldMethod isBreakpoint])
    ifTrue: [oldMethod client: method]
    ifFalse: [self addSelector: selector withMethod: method].
^selector
```

There are two major differences between this method and the compile:notifying:ifFail: method as defined in Behavior. First, this method saves the source code that was passed in and passes it along (using the sourceCode: message) to the CompiledMethodWithSource that's generated from the message send "methodNode generate." Also, the code checks to see whether the method being compiled used to have a breakpoint and, if so, preserves the breakpoint in the method dictionary. (This logic will be explained in detail in the next section.)

The final LightweightClass method is compilerClass, which simply returns a new class, LightweightCompiler, to be used when compiling lightweight class methods. Creating a new compiler class sounds overly ambitious, but it's actually quite simple, since the new class has only one method, newCodeStream; the rest are inherited straight from SmalltalkCompiler. This method is used to create a new CodeStream for use by the compiler. Since CodeStream generates CompiledMethods by default, we changed it to be parameterized by the kind of method generated, and so LightweightCompiler implements newCodeStream simply by returning a CodeStream that will generate instances of CompiledMethodWithSource. The implementation of CompiledMethodWithSource is just as simple. We changed three methods so that the sourceCode instance variable is interpreted as a source string (rather than a pointer to a file and offset), and the rest of its functionality is inherited from CompiledMethod.

With these few changes we now have an easy way to change the behavior of individual objects. We still need a good interface for doing that, though, and we'll describe our approach for that after first looking at breakpoints.

BREAKPOINTS

One of the typical things a programmer wants to do while debugging objects (and often in other debugging, as well) is to add "self halt" to a method—effectively adding a breakpoint. As it turns out, there's a simple way to add an initial breakpoint using the same technique that we used above with LightweightCompiler and CompiledMethodWithSource; we'll simply create a new class of compiled method, BreakpointMethod, and a compiler for generating instances of it. This variety of breakpoint has three advantages over the "self halt" version: They are easier to add and remove, since it's done by menu rather than by typing; they don't affect the various change mechanisms, so the change set and change log don't include trivial changes for adding (and presumably later removing) a halt in a method; and they are invisible in source code, so a programmer who is browsing or debugging a breakpointed method will see only the normally defined code—the breakpoint is invisible. The one disadvantage of our technique is that you can halt only at the start of a method, though our design may be adaptable to cover breakpoints throughout a method's body.

BreakpointMethod is a subclass of CompiledMethod with one instance variable, clientMethod. In addition, we added a new instance variable, agent, to CompiledMethod. When a breakpoint is set on an existing CompiledMethod, a new BreakpointMethod is created, and these two instance variables are changed so that the BreakpointMethod's clientMethod is the CompiledMethod, and the CompiledMethod's agent is the BreakpointMethod. The body of a BreakpointMethod is always the same: It's the expression "Notifier handleBreakpoint." Thus, when a BreakpointMethod is executed, this expression is evaluated, and Notifier responds by updating its stack, replacing the BreakpointMethod with its client—the original CompiledMethod—and opening a debugger with that method in the top context. In this way, the Break-

continued on page 24...

ParagraphEditorinitializeDispatchTable.
ParagraphEditorinitializeAdditionsToDispatchTable.

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

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

Execute the following code, customizing as appropriate:

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

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

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

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

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

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

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

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

CAVEATS

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

CONCLUSION

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

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

Acknowledgments

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

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

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

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

An Advanced Emergency Procedure for Objectworks

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

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

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

OTHER DEBUGGING AIDS

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

Isolate Debugging Code in a Subclass

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

Graphical Feedback

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

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

```
ScheduledControllers activeController sensor cursorPoint
```

You can position the cursor explicitly with:

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

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

```
Rectangle fromUser
```

Indicate an area with a filled rectangle:

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

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

```
Cursor sense
```

To translate to coordinates for a pane use:

```
Cursor sense mapScreenToClient: aPane
```

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

```
Cursor offset: aPoint
```

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

```
Display rectangleFromUser
```

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

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

Magic Debugging Keys

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

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

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

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

After compiling those methods, be sure to execute

Applications of Smalltalk in scientific and engineering computation

Richard L. Peskin

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

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

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

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

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

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

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

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

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

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

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

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

```
3 halt raisedTo: 2
```

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

Slowing Down the Action

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

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

For Smalltalk/V for OS/2:

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

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

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

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

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

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

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

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

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

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

HOW DO I GET OUT?

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

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

Quitting while You're Ahead

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

```
ObjectMemory quit
```

or

```
ObjectMemory quitPrimitive
```

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

In Smalltalk/V, the expression is:

```
Smalltalk exit
```

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

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

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

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

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

In Smalltalk/V for Windows, you can use the `WindowList`

Source Code for Blocks

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

```
aBlockClosure method getSource
aMethodContext sourceCode
```

For Smalltalk/V for OS/2:

```
aBlock homeContext method sourceString
```

Decompiling a Method in Objectworks

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

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

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

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

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

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

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

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

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

Entry Points

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

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

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

WHERE AM I GOING?

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

Breakpoints

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

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

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

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

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

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

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

For Smalltalk/V:

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


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

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

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

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

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

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

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

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

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

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

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

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

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

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

Identifying the Current Context

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

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

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

For Smalltalk/V:

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

Audible Feedback

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

```
Screen default ringBell.
```

For Smalltalk/V:

```
Terminal bell.
```

Catching It in the Act

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

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

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

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

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

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

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

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

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

Alternative to Walkbacks and Notifiers

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

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

For Smalltalk/V:

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

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

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

For Smalltalk/V:

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

...continued from page 1

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

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

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

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

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

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

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

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

Menu Hooks for Inspectors

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

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

Object identity

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

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

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

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

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

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

```
Smalltalk removeKey: #GlobalOne
```

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

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

Names

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

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

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

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

WHERE AM I AND HOW DID I GET HERE?

An object encapsulates both behavior and data. In addition to

THE BEST OF comp.lang.smalltalk

Alan Knight

Good code, bad hacks

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

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

THE BASICS

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

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

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

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

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

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

DOCUMENTATION

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

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

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

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

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

OTHER CRITERIA

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

Use Global Variables Sparingly

Bad—the use of global variables

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

Separate Domain and Interface

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

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

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

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

Avoid Long Methods

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

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

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

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

Avoid System Changes

Bad—making changes to system classes instead of subclassing

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

Keep Instance Creation Simple

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

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

```
new
    ^super new initialize
```

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

Use System Classes

Good—using system classes wherever possible

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

Work within the System

Good—using MVC, dependency mechanisms, and processes

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

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

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

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

CONCLUSION

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

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

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

CALENDAR

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

outputStream
    ^String new writeStream
```

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

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

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

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

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

commentCharacter
    ^$#
```

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

PATTERN: SEPARATE ABSTRACT FROM CONCRETE

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

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

Context

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

Problem

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

Constraints

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

Solution

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

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

Example

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

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


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

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

Choose Names Carefully

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

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

PUT CODE IN THE RIGHT PLACE

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

Place Code Well

> A series of messages sent to some object other than **self** is probably badly placed code. That series should be moved to the class of the receiver.
>
> Note: This is the hardest thing to do well in O-O programming, but it pays very high dividends when done well.

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

Avoid Checking Types Explicitly

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

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

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

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


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

than

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

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

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

Put Conditional Behavior in Subclasses

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

Introducing subclasses where other languages might use enumerated "type" variables is often good style. It is a problem if instances may change their type, but, otherwise, it can be very useful. In many ways, it's similar to the previous point. Instead of having conditional statements on the enumeration, we sim-

ply ask instances to perform some function. They will automatically do it in the appropriate way, and the language mechanisms will do the testing for free.

FAILURE MODES

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

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

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

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

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

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

CONCLUSION

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

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

SMALLTALK IDIOMS

Kent Beck

Inheritance: the rest of the story

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

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

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

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

PATTERN: COMPOSE METHODS

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

Context

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

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

Problem

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

Constraints

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

Solution

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

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

Example

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

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS MAIL PERMIT NO. 279 DENVILLE NJ

POSTAGE WILL BE PAID BY ADDRESSEE

The Smalltalk Report
Subscriber Services Dept SML
PO Box 3000
Denville NJ 07834-9821

BUSINESS REPLY MAIL
FIRST CLASS MAIL PERMIT NO. 4362 SAN JOSE, CA

POSTAGE WILL BE PAIDBY THE ADDRESSE

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

SERVIO CORPORATION
2085 HAMILTON AVENUE
SUITE 200
SAN JOSE, CA 95125-9985

The Smalltalk Report

☞ **Provides objective & authoritative coverage on language advances, usage tips, project management advice, A&D techniques, and insightful applications.**

"If you're programming in Smalltalk, you should be reading **The Smalltalk Report**"

❑ **Yes, I would like to subscribe to The Smalltalk Report** Date_____

❑ 1 year (9 issues)
- ❑ Domestic $69.00
- ❑ Foreign $94.00

❑ 2 year (18 issues)
- ❑ Domestic $128.00
- ❑ Foreign $178.00

Method of Payment
- ❑ Check enclosed (payable to **The Smalltalk Report**)
- ❑ Bill me
- ❑ Charge my: ❑ Visa ❑ Mastercard ❑ Amex

Card No. _____
Exp. Date _____
Signature _____

Name _____
Title _____
Company _____
Address _____
City _____
State _____
Zip _____
Country _____
Phone _____

1. **Which dialect of Smalltalk do you use:**
- ❑ Smalltalk V
- ❑ Smalltalk-80
- ❑ Other _____

2. **What is your involvement in software purchases for your department/firm:**
- ❑ Recommend Need
- ❑ Specify Product
- ❑ Make Purchase
- ❑ None

3. **Which operating system supports your software:**
- ❑ UNIX
- ❑ DOS
- ❑ OS/2
- ❑ Windows
- ❑ Other _____

4. **What is your company's primary business activity:**
- ❑ Computer/Software Development.
- ❑ Manufacturing
- ❑ Financial Services
- ❑ Government/Military/Utility
- ❑ Educational/Consulting
- ❑ Other _____

5. **For how long have you been using Smalltalk:**
- ❑ Less than one year
- ❑ 1–3 years
- ❑ 3+ years

E3GG

A member of the Object Marketing Network

fax to 212/274-0646

SIGS PUBLICATIONS

The Smalltalk Report

The International Newsletter for Smalltalk Programmers

September 1993 Volume 3 Number 1

BUILDING OBJECT-ORIENTED FRAMEWORKS

by Nik Boyd

Object system architects have long understood the value of frameworks. Frameworks provide a powerful way to organize and build interactive object systems. While classes define the structure and behavior of individual objects, frameworks define the structure and behavior of interactive object systems and subsystems (architectures). Just as classes provide leverage from the reuse of solutions to component problems, frameworks provide leverage from the reuse of solutions to systemic problems. Classes and frameworks complement each other for object modeling coordination.

Object system architects have sought ways to discover, describe, and define useful frameworks. This article explores some issues related to designing and building object systems, especially using frameworks. This article proposes that frameworks can be made first-class objects and describes the implementation of a Framework superclass for Smalltalk.

First-class frameworks provide a way to formalize the relationships between the objects in a system and factor out their patterns of interaction. Framework classes provide new opportunities for design, development, and reuse in object systems. They can be used to create very general or specialized event-driven systems. By making frameworks first-class objects, they derive and supply the same benefits as other objects: They can be built and reused with existing tools.

HOW THIS WORK EVOLVED

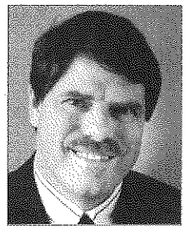
Smalltalk's browsers provide essential tools for quickly building and evolving objects. These tools organize and present objects and their definitions. The internal workings of these browsers can be quite complex. As a result, the classes that implement these browsers tend to have many methods.

The complexity of these browser classes contributes significantly to the difficulty of developing new tools for Smalltalk. This observation leads naturally to the following question: How can these browsers be broken down into more easily integrated and reusable components? The Model-View-Controller (MVC) framework[1] and its alternatives[2,3] provide great value, but do not completely resolve the problem of component integration.

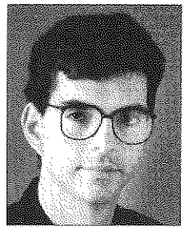
Early experiments with refactoring some new tools led to ways of loosely coupling their components using a kind of "smart" linkage. These component connections included their own behavior. After exploring some alternatives, it became obvious that these experiments had produced a way of implementing mediators.[4] Patterns began to emerge when the browser components were coupled together using mediators. This observation led naturally to the realization that some of these interaction patterns could be factored out and reused. Such refactoring created first-class framework objects whose behaviors are governed by interaction contracts.[5] Framework classes map interaction contracts directly onto inheritance hierarchies.

continued on page 4...

Contents:



John Pugh



Paul White

EDITORS' CORNER

t's been a busy spring and summer for conferences. Here are a few Smalltalk-related perspectives on those that one or the other of us has attended recently.

In May, Digitalk held their second conference for developers, DEVCON'93 in Costa Mesa, CA. The audience, which was populated by many representatives from banking and insurance companies, reflected very much the move of the MIS community into Smalltalk development. The conference program catered to this community with a heavy emphasis on the use of Smalltalk/V and PARTS in client-server computing. In one of the liveliest presentations, Amarjeet Garewal from the Bank of America described his firm's client-server development, ACA (A Cooperative Application). ACA facilitates distributed computing using Smalltalk and legacy systems, and was an award winner in the Object Applications category at the recent ObjectWorld conference. Watch for an upcoming article from Amarjeet in the REPORT. For Smalltalk aficionados who wanted to learn more of the "meta-world" of Smalltalk, Dave Smith from IBM give an inimitable reprise of his "Behavior of Behavior" presentation. SMALLTALK REPORT columnist Kent Beck dispelled a few Smalltalk myths and provided some invaluable insights into how to write high-performance Smalltalk programs. It was also Digitalk's 10th anniversary—they threw a good party!

June was the month for the large ObjectWorld conference in San Francisco. The Smalltalk story of note there was the demonstration of Hewlett-Packard's Distributed Smalltalk product—the first complete implementation of the Object Management Group's CORBA specification for distributed computing. Using Distributed Smalltalk, programmers can access distributed objects transparently without regard for whether the objects are local or remote. At the conference, users of Distributed Smalltalk in the HP booth were able to access objects residing in a Gemstone database in the Servio booth. Distributed Smalltalk consists of approximately 150 classes that sit on top of ParcPlace Systems' VisualWorks product. Watch out for upcoming articles on distributed computing with Smalltalk in future issues.

For the past few years, many people have been discussing the issue of frameworks as a mechanism for achieving reuse in object-oriented systems. For most, however, the issue of finding these frameworks is elusive, to say the least. As this month's lead article, Nik Boyd provides a description of how frameworks can be made first-class objects by introducing a Framework abstract class to Smalltalk and provides examples illustrating how best to use it.

Three of our columnists check in this month. Alan Knight addresses the issue we raised in our last editorial, namely making extensions to the base Smalltalk environment. In his column, he reports on "home-brewed" enhancements that have been posted to the Internet news group. In his column this month, Kent Beck continues his discussion of using inheritance effectively by introducing a pattern to be applied when attempting to make decisions concerning the factoring of subclasses. Greg Hendley and Eric Smith are back, describing how to take advantage of the object-dependents mechanism provided by Smalltalk when trying to keep multiple windows that are displaying inter-dependent information in sync. Finally, Dan Lesage reviews Dan Shafer's new book, SMALLTALK PROGRAMMING FOR WINDOWS.

Enjoy the issue—and welcome to our third year!

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
Adele Goldberg, ParcPlace Systems
Tom Love, IBM
Bertrand Meyer, ISE
Meilir Page-Jones, Wayland Systems
Sesha Pratap, CenterLine Software
Cliff Reeves, IBM
Bjarne Stroustrup, AT&T Bell Labs
Dave Thomas, Object Technology International

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

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

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

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

Circulation
Stephen W.Soule, Circulation Manager

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

Administration
David Chatterpaul, Accounting Manager
James Amenuvor, Bookkeeper
Margot Patrick, Assistant to the Publisher
Claire Johnston, Conference Manager
Cindy Baird, Conference Technical Manager

Margherita R. Monck
General Manager

SIGS PUBLICATIONS

Publishers of JOURNAL OF OBJECT-ORIENTED PROGRAMMING, OBJECT MAGAZINE, THE C++ REPORT, THE SMALLTALK REPORT, THE INTERNATIONAL OOP DIRECTORY, and THE X JOURNAL.

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 Inc., 588 Broadway, New York, NY 10012 212.274.0640.

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. For service on current subscriptions call 800.783.4903. Submit articles to the Editors at 509–885 Meadowlands Drive, Ottawa, Ontario K2C 3N2, Canada, 613.225.8812 (v), 613.225.5943 (f).
PRINTED IN THE UNITED STATES.

Highlights

Excerpts from industry publications

SOM

In practice, [IBM's] SOM (System Object Model) will allow programmers to "package" objects into blocks of code, of class libraries, that can be readily accessed from a C++ or Smalltalk program. Next month, IBM will extend SOM with a full CORBA (Common Object Request Broker Architecture) model. This Distributed SOM, or DSOM, spec will let objects be transparently accessed either locally or across a network.

IBM reveals its new software 'object'-ive, Alexander Wolfe, ELECTRONIC ENGINEERING TIMES, 5/17/93

POINTER-SAFE

At least triggers are specified in an SQL variant. SQL has no pointers and there is no need to worry about wild stores. Even if the application is written in a language that is not pointer safe (e.g., C) a wild pointer or running off the end of an array will not corrupt the database. However, most object database vendors and at least one relational vendor allow behavior specified in C or C++ to be optionally linked into a server process, and server processes contain very large caches of data. The problem in the relational environment is that the rows in the cache are assumed to satisfy all integrity constraints and that the cache is often shared amongst multiple clients. A seemingly experienced application developer once told me, in all seriousness, that mature C code doesn't produce any wild stores (and you wonder why DBAs sometimes seem paranoid). A wild store in this scenario can result in corrupted data being committed to the database. And the corrupted data might not have been read by the offending application program. Many object databases have the same problem with behavior specified in C or C++. These databases tend to bulk copy their caches to disk at transaction commit. This is one of the major reasons why I have always believed that a pointer-safe language such as Smalltalk is a much better data-manipulation language that C or even C++.

ODBMS: Tear down the walls, Jacob Stein, OBJECT MAGAZINE, 7–8/93

Shafer's style of writing in this book is down to earth. This should appeal to new programmers, but there are instances where I found the style to be a little subterranean. On page 141, for example, Shafer writes:

(It's amazing to think one can actually get paid for doing this kind of work, isn't it?)

I hope I never accidentally put that into my application comments!

On the plus side, this book has really made strides in the area of integrating an application into its surroundings. Appendix B discusses DDE and DLL interfaces and provides an example of adding a DDE link to the Calendar application from Microsoft Excel. The DLL example shows how to use multimedia extensions in combination with a sound board. The example demonstrates how to modify the Calendar project to play a sound file instead of beeping for alarm events.

The end result of all these enhancements gives a Calendar application that is comparable in function to the Microsoft Windows desktop calendar. I believe that most programmers would classify this to be a true application, albeit a simple one.

Shafer demonstrates the use of fast prototyping as a mechanism for building applications. Throughout the book, he proposes designs that have minor flaws contained within them. He then leads the reader through the analysis required to correct the problem. This highlights an important point pertaining to the design of graphical applications. Most of the *discovered* problems have to do with event handling, sequencing, bad initialization and proper notification of change. To further complicate the analysis, these problems occur within a multi-window, multi-pane, multi-widget environment. This is also true in the real world: The hard part is not defining the visual aspects, it is getting the glue right. This book does an excellent job in highlighting these kinds of problems and demonstrating the type of analysis is required to correct them.

Once you overcome the silly book cover, the cartoons on the back and the fact that the publisher's name is about 3 times the size of the author's, the content of this book will be very useful to new Smalltalk programmers. The calendar application can form the basis of an introductory Smalltalk course. I know of one company that has modeled part of its internal training examples on those presented in the book. This book is a colossal improvement over its predecessor and it demonstrates what it takes to start building applications under Windows using Smalltalk. I recommend this book to new Smalltalk programmers who wish to quickly develop small scale applications within the Windows environment. ▣

Dan Lesage is responsible for Distributed Systems Frameworks at Object Technology International Inc. This means that he gets to act as trial arbiter between very unlike pieces of hardware and software, protocol arbiter between collaborating classes in frameworks, personnel arbiter between team members and aqueous medium arbiter between aggressive piscatorial members of his aquaria. It occasionally means that he gets to develop software in Smalltalk. He can be reached at 613.820.1200 or dan@oti.on.ca.

BOOK REVIEW

by Dan Lesage

SMALLTALK PROGRAMMING FOR WINDOWS

by Dan Shafer with Scott Herndon and Laurence Rozier

Prima Publishing
Roclin, CA
phone: 916.786.0426
fax: 916.786.0488
$39.95
ISBN 1-55958-237-5 1993

I am waiting for the day of the truly paperless book. The day when reading on an electro-luminescent or photo-polarized device provides me with as little eye strain as reading flat paper. I am sure that Dan Shafer is waiting for this day as well. On that day, the problem of publishing a timely technical book about rapidly changing technology will no longer exist.

Eighteen months ago, I reviewed Shafer's original Smalltalk book, entitled PRACTICAL SMALLTALK (THE SMALLTALK REPORT, October 1991). One of the issues I raised in that review was that the book presented examples in Smalltalk/V 286, just when Digitalk was moving toward PC desktop integration with Windows and OS/2 Presentation Manager. The paradigm used for modeling these new user interfaces had changed drastically from Model-Pane-Dispatcher. MPD lost its sex appeal for solving UI problems, although the fundamentals of Smalltalk were the same. Real-world Smalltalk development had moved on to a different paradigm.

Shafer's new book, which uses V Windows 2.0 as its base, is more timely than its predecessor. However, it is interesting that Digitalk's focus has moved onto Parts, once again leaving Shafer to play catch-up. What we need is the ability to publish a book directly from a Smalltalk image!

Once again the focus of the new book is a practical introductory guide for novice Smalltalk users. It acts as a supplement to the material provided by Digitalk. The format of the book is similar to the previous one. After two introductory chapters, it leads the reader through chapter pairs. The first chapter of each pair introduces important Smalltalk classes. The second of the pair highlights the use of these classes within a working example application.

The book describes seven detailed projects. The first is a List Prioritizer that prompts the user to prioritize text entries. The second consists of a Counter widget that introduces interaction between subpanes. The third project is a Calendar application that displays monthly pages, allowing you to navigate dates, highlighting holidays and the current date. The fourth application is an Appointment Book built by extending the calendar application in the third project. The Appointment Book introduces the ViewManager class. The fourth project also demonstrates how to manage multiple window interaction by adding a text based appointment window to the calendar. The fifth project is a Bar Graph Editor and Viewer. The sixth consists of a Form Designer that demonstrates how to create a user interface layout from a Smalltalk outline. The last project consists of a Clock that also hooks into the Calendar application. The clock is responsible for displaying the time and sounding alarms and chimes. The Clock project demonstrates the multiprocessing capability built into Smalltalk and how to use it in combination with ViewManager.

I found that the example projects contained within the book had greater relevancy to developing real applications than the ones presented in PRACTICAL SMALLTALK. Only the List Prioritizer, Counter, and Bar Graph Viewer appear as upgraded versions of examples used in the previous book. The remaining projects simulate the process of building real applications. They require the developer to add new functions to existing software rather than create designs from scratch. Changing the Calendar viewer into a time-based Appointment Book typifies how Smalltalk developers must constantly reorganize their code to accomodate new requirements. The Clock project, which is the cumulative effect of these requirements, provides new Smalltalk programmers with insight into the power of classes such as Time, Processor and Context (blocks). This last project demonstrates how to make these classes collaborate to simulate the behavior being modeled. The result of completing the last project is a sense of satisfaction and confidence. Developers should feel comfortable browsing the class hierarchy as they develop more complex applications.

The book includes a 3.5-inch diskette that contains V Windows 2.0 code, so browsing the examples is easy. Just remember to remove the diskette immediately when you buy the book or you will find that after a while, the soft back cover will look like it has been run over by an office chair!

There appear to be some errors within the printed Smalltalk code that do not appear on the diskettes. Pages 184 through 186 contain numerous syntax errors and erroneously repeated code. Unless you are a masochist, you should browse the code from your image rather than read the book to ensure correctness. Of course, that means you need your *paperless* book again, as you fly from Boston to Ottawa. Hmmm...

...continued from page 1

This article describes the results of these experiments: The role that frameworks can play in system design, and how framework classes can be used to define the structure and coordinate the behavior of objects in systems. We begin by exploring some issues related to object design and system design.

OBJECT DESIGN AND SYSTEM DESIGN

We often solve large problems by breaking them up into smaller problems and combining the solutions (divide, understand, integrate: *solve e coagula).* Just so, we can divide large systems of interacting objects into smaller collaborations, or subsystems. This allows us to better understand and manage the structure and behavior of the larger system.

Two key concerns of object system architects are the *right factoring* of behavior and the *right coupling* of objects. Although different aspects of a design, factoring and coupling decisions often influence each other. For example, creating a new object class presents a question that arises frequently in object system design: Where does the new class belong in a class hierarchy? This critical design activity incorporates both factoring and coupling decisions because objects serve as the essential unit for both factoring and coupling in object systems.

The class location decision can be made easier by looking at the proposed service responsibilities of the new class and asking some questions. Does the new class provide the same (or substantially similar) services when compared to another existing class? Does it add new services or change the implementation of some services? Does it remove any services? When a new class shares (and perhaps adds to) the public interface of an existing class, the new class is a good candidate for subclassing the existing class. When the public interface of the new class is not substantially similar, but needs the services of an existing class, the new class should be a client of the existing class. When a new class shares some portion of the public interface of an existing class, the hierarchy may need to be revised, splitting out the shared interface into a new, more general superclass shared by both the existing and newer subclasses. Finding the best location for object behaviors is the essence of right factoring.

RIGHT FACTORING

Factoring characterizes how well responsibility for services are distributed throughout an object system or class hierarchy. Ideally, each unique piece or pattern of behavior has a unique location within each object system or class hierarchy.

Classes may be organized initially based on data and the operations on that data. However, classes should finally be organized based on their service responsibilities and collaborations. Each object in a system is assigned responsibility for providing certain services to its clients. Responsibility-based design (RBD) takes the client/server approach to its logical conclusion in the design of finegrained objects and collaborative subsystems.[6-9]

Many experienced object designers have suggested that good class hierarchies tend to be *deep and narrow.* A hierarchy is considered deep when there are many intermediate superclasses between the most specialized classes and the top of the hierarchy. A hierarchy is considered narrow when each class in the hierarchy adds relatively few public services.

Class libraries tend to evolve over time until they become stable and mature. However, we must be careful if we don't want such stability to mean that they ossify! This can happen in large systems when a few basic objects are used repeatedly, creating many dependencies. The stability created by such dependencies may argue against redesign, creating a kind of inertia.

Early design evolution should be encouraged in order to prevent premature stability. Object modeling[10] can help to accelerate the process of evolution during class and system design. Design iteration provides opportunities for revisiting and revising object and system designs through *refactoring.*[11]

Refactoring applies one or more kinds of behavior preserving transformation to an object model. The behavior of the modeled objects is redistributed so that they are simpler and provide better opportunities for reuse. Even fairly stable class hierarchies may be improved by subjecting them to refactoring.[12]

One frequently used example of refactoring is generalization. When two or more subclasses share some common behavior, a new more general superclass can be created by factoring out the shared behavior.

Many of the transformations permitted by refactoring can be automated. Automating the refactoring process could eventually lead to the development of a kind of "lint" eliminator for object designs.

RIGHT COUPLING

Coupling characterizes the relative visibility and independence of objects in relation to each other. Ideally, objects and classes should only be visible to those clients that need to see them.

When one object depends implicitly on another, they are *tightly coupled.* Object instances are tightly coupled to their classes. When one object depends directly on the visibility of another, they are *closely coupled.* Smalltalk instance, class, and pool variables are are closely coupled to the instances that reference them.

When one object references another only indirectly through an opaque reference or through some accessing or structural traversing message(s), it depends only on some portion of the other's public interface and may be *loosely coupled.* Table 1 summarizes the relationships between visibility and coupling.

Thus, appropriate visibility is essential for achieving right coupling. Often, the success of a large programming project hinges on right coupling. Right coupling can only be achieved if the system architect has an awareness of coupling and visibility

Table 1. Relationships between visibility and coupling.

Visibility	Coupling
Implicit	Tight
Immediate	Close
Opaque	Loose
None	None

forms an expression of the form self changed: attribute. The parameter attribute varies depending on just what part of the domain model object was altered. For the change of name example, this argument would likely be *name.* In such a case, the setter method for name in the class Customer might look like the following:

```
Customer>>name: aString
        "Accessing -- Set my name. Update anybody who's interested."
    name := aString.
    self changed: name
```

5. Whenever an object is sent the changed: message, as in event 4, all other objects which have been registered as dependents on the receiver of the changed: message receive update: messages. The argument passed along with the update: message is the same as that passed in with the original changed: message which started the process.

6. In processing the update: message, the application control compares the argument with those identifying aspects of the domain model in which it is interested. If a match is found, then the associated interface object is informed that some of the data it is displaying is no longer valid and must be updated. This is done by sending the interface object a message that tells it just what data needs to be redisplayed. If this were a view on the Customer as in the preceding examples, the method for update: would look, in part, like the following.

```
CustomerEditorControl>>update: aspect
        "Updating -- Some part of my domainModel has changed. See if
        it is a part in which I'm interested. If it is, then direct
        the userInterface to update it."

    aspect = = #name
        iffrue: [^ self userInterface invalidateName].
    aspect = = #company
        iffrue: [^ self userInterface invalidateCompany].
    ^ super update: aspect
```

7. As Control B is also a dependent of Domain Object it will also receive an update: message of the same form as that received by Control A in event 5. This provides application B with an opportunity to keep its view of the domain object up to date even though application A was the source of the change. Application B does not need to know the source of the change. All it needs to know is what change took place. This update: message provides it with this information. The two views of Domain Object remain in sync.

8. Control B will handle the update: notification in much the same way as did Control A in event 6. In fact, if these are the same kind of views of Domain Object, then it will handle the message in exactly the same way. The end result is that a message will be passed on to Window B telling it that it must refresh the display of the changed item.

After the list of dependents of Domain Object is exhausted, that is each member of that list has received and processed the update: message, the process of changing an attribute of the domain model object is complete. Only at this point does the processing of the cmdSetAttribute message from event 2 complete.

Note that the domain model object did not need to know much about the application to provide this notification. All it needed to know is when to yell, "I've changed!" Other objects may or may not be interested. If they're not interested, they just won't listen.

> "Objects may or may not be interested. If they're not interested, they just won't listen."

CLEAN UP

When any of these windows are shut down, the dependency links with the domain model objects must be broken. This is best done using the removeDependent: message. When a window is closed it must, before it goes away entirely, pass on to its control object a message allowing it to clean up as well. A message like cleanUp will do nicely:

```
CustomerEditorModel>>cleanUp
        "I'm about to be terminated, clean up
        any messes I've left laying about."
    self domainModel removeDependent: self
```

The Object Dependents mechanism can be particularly useful for keeping collections of information up to date dynamically. This will be the topic of a future column. ▣

Greg Hendley is a member of the technical staff at Knowledge Systems Corporation. His specialty is custom graphical user interfaces using various dialects of Smalltalk and various image generators. Eric Smith is also a member of the technical staff at Knowledge Systems Corporation. His speciality is custom graphical user interfaces using Smalltalk (various dialects) and C. The authors may be contacted at Knowledge Systems Corporation, 114 MacKenan Drive, Cary, NC 27511, 919.481.4000.

> Whenever some aspect of Domain Object that might be of some importance to the outside world changes, the method of the domain object that actually changes the value performs an expression of the form self changed: attribute.

update message
update: sender
update: arg()
update: arg() with: arg1
update: arg() with: arg1 with arg2

An object, A, may register itself as a dependent on another object, B, by sending B the message addDependent: with itself, A, as the argument. All dependents of an object are removed by sending the object the message release.

A NOTE FOR DIGITALK USERS

Digitalk does not provide one method that is very useful in dealing with Object Dependents. The missing method is Object>>removeDependent: and a possible implementation is:

```
Object>>removeDependent: aDependent
        "Remove a single object from my list of dependents."

    | dependents |

    (dependents := Dependents at: self ifAbsent: [^ nil])
                    remove: aDependent
                    ifAbsent: [].

    dependents isEmpty ifTrue: [self release]
```

Digitalk users should also beware of the confusion possible because ViewManagers implement their own independent changed-update framework, which is unrelated to Object Dependents though it uses much the same protocol. To avoid problems, we won't be sending changed messages to view managers.

TWO VIEWS ON ONE OBJECT

To keep all of the windows on a particular domain model object current, the domain model objects will generate self changed: messages whenever some aspect of their state has changed. It is assumed that when an view is opened on any dynamically updatable domain object, that the application control object registers itself as a dependent of the domain model object it is representing to the user. This will insure that the application control will receive the update: message when the state of the domain model object changes. It is also assumed that the responsibility for undoing the dependency link when the window is closed also resides with the application control object.

SETTING UP

When a window is opened on a domain object, using an openOn: message for example, the window informs its application control object that this domain object is to be its model object. It is at this time that the application control object should register itself as a dependent of the domain model. The following methods illustrate this set up:

```
CustomerEditor>>openOn: aCustomer
        "Scheduling — Open myself up as a window
        on the given Customer."

    self control domainModel: aCustomer.
    self open


CustomerEditorControl>>domainModel: aCustomer
        "Accessing — Set my reference to my domain model object.
        Make myself a dependent of this object."

    domainModel notNil ifTrue: [domainModel removeDependent: self].
    aCustomer notNil ifTrue: [aCustomer addDependent: self].
    domainModel := aCustomer
```

Given this set up, Figure 2 provides an illustration of a generic scenario for what happens when some attribute of a displayed domain object is changed by the user in one of two views on that object. In this example Control A and Control B are both dependents of Domain Object.

1. The user uses some control in the window to alter the value of an attribute of the domain object being presented to him. For example, a the name of a Customer may be changed.
2. As a result of manipulating a control, a command message is forwarded to the application control object of the window the user is working with. In the case of changing the customer's name, this might be a message like cmdSetCustomerName:.
3. In the course of processing the command message, Control A sends a message to the Domain Object to inform it that it must change some of its internal state. To continue the customer name example, this would likely involve sending Domain Object the message name:.
4. Whenever some aspect of Domain Object that might be of some importance to the outside world changes, the method of the domain object that actually changes the value per

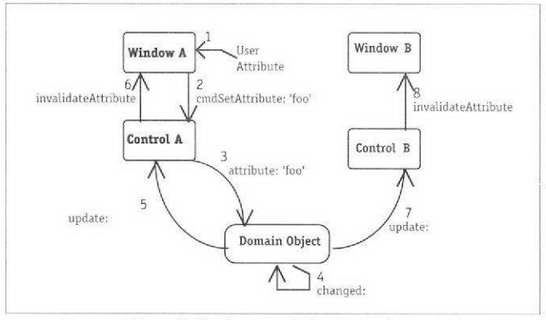


Figure 2. Keeping two windows up to date.

issues, and has tools that provide him with real options for dealing with those issues.

Component classes, module classes,[13] and framework classes complement one another in controlling coupling and visibility in Smalltalk systems. They also provide complementary mechanisms for factoring. The issues raised regarding the factoring of behavior and the coupling of objects can be dealt with formally by designing objects using contracts.

DESIGNING WITH CONTRACTS

Contracts are design abstractions. They provide high-level descriptions of:

- The behavior (and structure) of a component object
- The collaborations between the components that form a subsystem
- The interactions between the participants in a framework.

Classes define the service capabilities of their instances. These services can be organized using *protocols.* Protocols are generally used to represent the contracts provided by objects. Protocols generally characterize the services they organize using descriptions derived from verb phrases such as *initializing-releasing* (instances), *accessing* (some state information), *computing* (some value).

Sometimes a complex set of related services can best be implemented and simplified by assigning responsibility for some contract(s) to a separate class. The set of resulting classes can then be organized as collaborators in a subsystem. Responsibility-based design[9] can be used when defining and refining the contracts fulfilled by components and subsystems.

In Smalltalk, module classes[13] can be used to organize and provide opaque access to subsystems. Like component classes, module classes can be instantiated. Whether through the module class or one its instances, each module serves as a gateway, providing access to the services of its internal subsystem.

Interaction-oriented design can be used when defining and refining the interaction contracts fulfilled by frameworks. In interaction-oriented design, the interactions between objects are first-class entities in the design space.[5] Using framework classes, these first-class designs can be implementated as first-class objects.

THE FRAMEWORK SUPERCLASS

Listings 1 and 2 provide the Smalltalk source code that implements the Framework superclass. The Framework superclass is intended to be subclassed to create both general and specialized frameworks. The Framework superclass is responsible for providing the following services:

- Building a framework from participants
- Resolving roles for participants
- Defining roles and their responsibilities
- Validating participants for roles
- Translating events into messages

When a framework instance is built, some of the participants are components, but some may be other frameworks. These nested frameworks are given special treatment during the assembly of the framework in which they are embedded. Each nested framework is checked for unresolved roles. If any unresolved roles are found, they are filled using participants from the embedding framework by matching their role names. Thus, naming the roles and participants in a network of frameworks is an important activity.

This feature allows system architects to design and build networks of interlocked frameworks. Small frameworks and their components can be integrated so that events propagate through the network to produce the overall behavior of a large system.

Within a framework, each object has a role and must supply certain services in order to fulfill that role. An interaction contract defines the responsibilities of the objects that form a behavioral composition. The services each object must render in order to participate in a role may be defined explicitly as part of a framework class. When these specifications are defined for the roles of a framework class, they are verified when each instance of the framework is assembled.

Although framework role validation is feasible within any language system, it is easiest to implement when the language supports reflection directly. Reflection provides objects with access to information regarding their own behavior. Sometimes this language feature is described as object self-knowledge. Smalltalk is one of the few commercial languages that support reflection.

The use of reflection by framework classes for validating role participants presents an interesting opportunity. This reflective information can be used to support the intelligent assembly of object frameworks. In Listing 1, the #assembleAs: method shows how the Collection class may be extended to support framework assembly from anonymous participants.

If the service requirements defined for each role differ sufficiently, they may be used to identify the role players needed from a collection of anonymous participants. Each anonymous participant can be examined to determine its most likely role within a framework based on the service requirements of each role. Once the roles of all the participants have been identified, the framework can be built without any need to explicitly specify their roles.

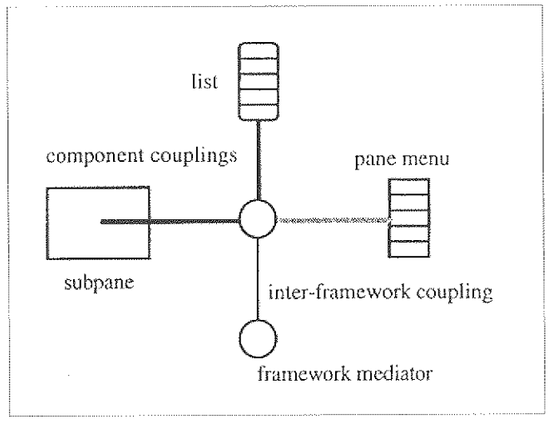


Figure 1. Key diagram.

EVENT NOTIFICATION AND TRANSLATION

The MVC framework and other similar ones typically broadcast event and change notifications to dependents. While this may be sufficient for simple frameworks, more complex frameworks need something more: the ability to target specific framework participants for event or change notification. For this reason the Framework class supports both kinds of notification mechanisms:

```
self notify: #someParticipant
  that: #somethingHappened.

self someParticipant
  notifyThat: #somethingHappened.
```

The #notify:that: request extends the Object class to provide event notification targeted at specific named dependents. The #notifyThat: request extends the Object class to provide broadcasting of events to all dependents (see Listing 1). The Framework class overrides #notify:that: to support targeting specific named participants. It also overrides #notifyThat: to translate events into actions.

SOME EXAMPLE FRAMEWORKS

The first two examples are described in Reference 5. Listing 3 shows a framework class that captures the SubjectView contract. The SubjectView contract manages a collection of views so that they all reflect the current value of a subject. By factoring out the behavior related to the contract into a separate framework class, the services that the subject and view classes must support are drastically reduced. This factoring allows these classes to be simplified to their essential behavior without concern for how they are used in a broader context.

Listing 4 shows how ButtonGroup, a specialization of the SubjectView contract, can be captured as a framework subclass. The ButtonGroup shows which button of a group of radio buttons is selected. Here again, the behavior required of the Button class is reduced, eliminating its need to retain any framework specific behavior.

The next example is derived from efforts to refactor some browser classes. A brief overview will suggest how such refactoring may proceed. The Framework superclass is subclassed by a hierarchy that supports the redirection and translation of the SubPane events used in Smalltalk/V. The class SubPaneMediator guides the interactions between one of the SubPane subclasses (i.e., Button) and some other component(s).

The component used by these mediators in addition to the subpanes is a SelectionList. The SelectionList class remembers the selection of a single item from a list of items. The item list may be either an IndexedCollection or an OrderedDictionary. The selection index of the list is either an ordinal number or an or-

continued on page 14...

GUIs

Greg Hendley & Eric Smith

Keeping multiple views up-to-date

In many Smalltalk applications, it is possible for the end-user to have several independent windows providing views of the same information (Figure 1). There may be several instances of one kind of window or parent and child windows that, though very different in appearance, share some overlap in the information they present. To prevent inconsistencies between windows, the changed-update (also known as *Object Dependents*) mechanism can be used to insure that all views the end-user has opened on a particular object are kept up-to-date with that object's most recent idea of what it looks like.

For example, if the user has a view of a Customer object that he opened directly and another view of the same Customer that was opened as a consequence of browsing a ServiceAgreement object, any changes make to one view of the Customer should be immediately reflected in the other. The user should not see two different views and be left to figure out which one is the most current.

BACKGROUND

The application architecture outlined in previous columns (THE SMALLTALK REPORT, May 1992 and October 1992) will be employed here. For those who have not yet been exposed to the Interface-Control-Model architecture, a brief glossary of terms is provided here.

- *Interface.* The component of the user interface whose job it is to present information to the end-user and accept input events from same. The interface translates user input to semantic actions such as mouse-clicks to selection or menu selections to commands. The interface has very little knowledge of the structure of the application of which it is a part. It has virtually no knowledge of the domain model (see below).
- *Control.* The control layer of an application is the component that understands the semantics of the application as a whole. This is where commands identified by the interface are actually carried out. The application control understands the relationships among the various domain model objects it works with. It also knows about the consequences of commands. This is the point where all the "brains" of the application (as the end-user sees it) reside.
- *Model.* The model is the meat of the system. This is where the real information is modeled (hence the name). If we were working with a circuit design application, this layer is where objects such as Circuit, Transistor, Diode, etc. would be found. These objects have only the most limited understanding that there is a user interface above them. They have no direct knowledge of user interface issues.

OBJECT DEPENDENTS

Both major dialects of Smalltalk provide essentially the same Object Dependents facility. The idea is that a client object, which wants to be informed when some other object changes, registers itself as a *dependent* of that object. Since the requisite behavior for maintaining dependencies is implemented in the class Object, all objects may have dependents, be dependent on other objects, or both.

The detailed operation of Object Dependents is a topic for another time. We'll have to be satisfied with just a quick look at the top level of the behavior. In the simplest terms, an object which has changed and may have dependents sends itself a *changed* message. This results in each of the dependents, if any, of the object in question being sent a matching *update* message. A list of possible changed messages and their matching update messages is presented below:

```
changed message
changed: arg()
changed: arg() with: arg1
changed: arg() with:arg1 with: arg2
```

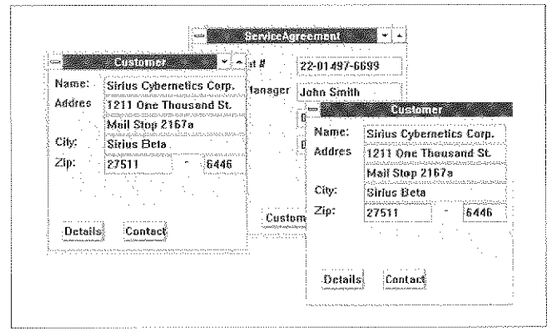


Figure 1. Two windows on a single Customer.

```
selectItem: item
  self selectionList selectItem: item.!

selectObject: selection
  self selectionList select: selection.! !

ListItemChooser subclass: #ListViewer
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: '' !

!ListViewer class methodsFor: 'validating roles' !

widget
  ^ #(
        deselect
        restoreWithRefresh:
        selection:
        selection
  )! !

!ListViewer methodsFor: 'translating events' !

deselected
  self widget deselect.!

listChanged
  self widget restoreWithRefresh: self selectedItem.!

selected
  self selectIndex: self widget selection.! !

!ListViewer methodsFor: 'changing component state' !

showSelection
  self widget selection: self selectedIndex.! !

ListItemChooser subclass: #ListButton
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: '' !

!ListButton class methodFor: 'validating roles' !

widget
  ^super widget, #( contents: )! !

!ListButton methodsFor: 'translating events' !

clicked
  self listSelections size > 1 ifTrue: [
        self widget contents: self nextSelection ].!

listChanged
  self selectionChanged.!

selectionChanged
  self widget contents: self selectedItem.! !

ListButton subclass: #MenuButton
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: '' !

!MenuButton class methodsFor: 'validating roles' !

selectionList
  ^super selectionList, #( popUpItems )! !

!MenuButton methodsFor: 'changing component state' !

nextSelection
  ^self selectionList popUpItems! !

ListButton subclass: #ToggleButton
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: '' !

!ToggleButton class methodsFor: 'validating roles' !

selectionList
  ^super selectionList, #( selectNext )! !

!ToggleButton methodsFor: 'changing component state' !

nextSelection
  ^self selectionList selectNext! !
```

SMALLTALK IDIOMS

Kent Beck

Inheritance: the rest of the story

In the June issue where I took on accessor methods, I stated that there was no such thing as a truly private message. I got a message from Nikolas Boyd reminding me that he had written an earlier article describing exactly how to implement really truly private methods. One response I made was that until all the vendors ship systems that provide method privacy, Smalltalk cannot be said to have it. Another is that I'm not sure I'd use it even if I had it. It seems like some of my best "reuse moments" occur when I find a supposedly private method in a server that does exactly what I want. I don't yet have the wisdom to separate public from private with any certainty.

On a different note, I've been thinking about the importance of bad style. In this column, I always try to focus on good style, but in my programming there are at least two phases of project development where maintaining the best possible style is the farthest thing from my mind. When I am trying to get some code up and running I often deliberately ignore good style, figuring that as soon as I have everything running I can simply apply my patterns to the code to get well-structured code that does the same thing. Second, when I am about to ship a system I often violate good style to limit the number of objects I have to change to fix a bug.

What got me thinking about this was a recent visit I made to Intelliware in Toronto. Turns out Intelliware is two very bright but fairly green Smalltalkers, Greg Betty and Bruno Schmidt (he's not nearly as German as his name). They hired me to spend two days going over the code they had written for a manufacturing application. The wonderful thing was, they had made every mistake in the book. It's no reflection on their intelligence; everyone makes the same mistakes at first.

What made their boo-boos so neat was that I was able to go in and, in two days, teach them a host of the most advanced Smalltalk techniques just by showing them how to correct errors. I'd say, "Oh, look, an isKind0f:. Here's how you can get rid of that and make your program better at the same time." Because I had a concrete context in which to make my observations, they could learn what I was teaching both in the concrete ("Yes, that does clean up the design") and the abstract ("Oh, I see. I can do that any time I would have used isKind0f:").

So, go ahead. Use isKind0f:. Use class == and == nil. Access variables directly. Use perform: a lot. Send a message to get an object that you send a message to. Just don't do any of these things for long. Make a pact with yourself that you won't stand up from your chair (or go to bed, or ship the system, or go to your grave. . .) without cleaning up first.

Some people are smart enough to write clean code the first time. At least, that's what they tell me. Me, I can't do that. I write it wrong, and then fix it. Hey, it's not like we're writing in C++ and it takes an hour to compile and link our programs. You may as well be making your design decisions based on code that works. Otherwise, you can spend forever speculating about what the *right* way to code something might be.

PATTERN: FACTOR A SUPERCLASS

As an alternative to the Separate Abstract from Concrete pattern, I'd like to present the way Ward Cunningham taught me to make inheritance decisions. It is very much in keeping with what I wrote above about letting your "mistakes" teach you the "right" thing to do. When you are programming like this, it feels like the program itself is teaching you what to do as you go along.

CONTEXT

You have developed two classes which share some of the same methods. You have gotten tired of copying methods from one to the other, or you have noticed yourself updating methods in both in parallel.

PROBLEM

How can you factor classes into inheritance hierarchies that share the most code? (Note that some people will say that this isn't the problem that inheritance should be solving. You wouldn't use this pattern if that was your view of inheritance.)

CONSTRAINTS

You'd like to start using inheritance as soon as possible. If you're using inheritance you can often program faster because you aren't forever copying code from one class to another (what Sam Adams calls "rape and paste reuse"). Also, if you are using inheritance, you don't run the risk of a multiple update problem, where you have two identical methods, and you change one but not the other. Ideally, for this constraint, you'd like to design your inheritance hierarchy before you ever wrote a line of code.

On the other hand, designed inheritance hierarchies (as opposed to derived inheritance hierarchies) are seldom right. In


```
!SubPaneMediator methodsFor: 'binding components' !

supportedEventHandlers
  ^ #(
        clicked:
        doubleClickSelect:
        getContents:
        getMenu:
        getPopupMenu:
        select:
  )!

handlerFor: event
  ^self supportedEventHandlers detect:
        [ :evh | event = (evh copyWithout: ($:) ) ]
        ifNone: [ nil ]!

support: event for: subPane
  | selector |
  selector := self handlerFor: event.
  selector isNil ifTrue: [ ^self ].
  subPane when: event perform: selector.!

supportEventsFor: subPane
  subPane class supportedEvents do: [ :event |
        self support: event for: subPane ].!

claimOwnershipOf: subPane
  subPane ifUnderstood: #supportedEvents do: [
        subPane owner: self.
        self supportEventsFor: subPane ].!

for: partName use: anObject
  | selector |
  super for: partName use: anObject.
  self claimOwnershipOf: anObject. "if SubPane"! !

!SubPaneMediator methodsFor: 'handling events' !

clicked: subPane
  self notifyThat: #clicked.!

doubleClickSelect: subPane
  self notifyThat: #doubleClicked.!

getContents: subPane
  self ifUnderstood: #getContents do: [
        subPane contents: self getContents ].!

getMenu: subPane
  self ifUnderstood: #getMenu do: [
        subPane setMenu: self getMenu ].!

getPopupMenu: subPane
  self ifUnderstood: #getPopupMenu do: [
        subPane setPopupMenu: self getPopupMenu ].!

select: subPane
  self notifyThat: #selected.! !
```

```
SubPaneMediator subclass: #ListItemChooser
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: '' !

!ListItemChooser class methodsFor: 'validating roles' !

roleNames
  ^ #( selectionList widget )!

selectionList
  ^ #(
        list:
        items
        selections
        selectIndex:
        selectedIndex
        selectItem:
        selectedItem
        select:
        selection
  )! !

!ListItemChooser methodsFor: 'accessing component states' !

getContents
  ^self listItems!

listItems
  ^self selectionList items!

listSelections
  ^self selectionList selections!

selectedIndex
  ^self selectionList selectedIndex!

selectedItem
  ^self selectionList selectedItem!

selection
  ^self selectionList selection!

selectionList
  ^self partnerNamed: #selectionList!

widget
  ^self partnerNamed: #widget! !

!ListItemChooser methodsFor: 'changing component state' !

changeList
  self selectionList list: self getList.
  "note: getList should be implemented by subclass
  method or prototype block"!

selectIndex: index
  self selectionList selectIndex: index.!
```

```
    supers do: [ :s |
        methodNames addAll: s methodDictionary keys ].
    methodNames := methodNames select: [ :n |
        n last == ($:) ].
    methodNames := methodNames collect: [ :n |
        n copyWithout: ($:) ].
    ^methodNames!

doesNotUnderstand: aMessage
"Try handling aMessage, assuming it is accessing the parts of the
receiver. If the part accessed is a block context, answer the result of
evaluating the block with the receiver and arguments from aMessage as
arguments. Otherwise, answer the accessed part. If aMessage does not
access a part, let the superclass handle aMessage."
    | part |
    (parts respondsTo: aMessage selector) ifFalse: [
        ^super doesNotUnderstand: aMessage ].
    part := self partNamed: aMessage selector ifNone: [
        ^parts perform: aMessage selector
                withArguments: aMessage arguments ].
    part isContext ifTrue: [
        aMessage arguments isEmpty ifTrue: [
                ^part value: aMessage receiver ].
        ^part value: aMessage receiver
                value: aMessage arguments ].
    ^part! !
```

Listing 3: SubjectView Contract.

```
"The following example is derived from the contract SubjectView
described on page 171 of [HHG90]."

!SubjectView class methodsFor: 'validating roles' !

roleNames
  ^ #( subject view )!

subject
  ^ #( value value: )!

view
  ^ #( showValue: )! !

!SubjectView methodsFor: 'supporting subject' !

setValue: value
  self getValue = value ifTrue: [ ^self ].
  self subject value: value.
  self notify.!

getValue
  ^self subject value!

notify
  self views do: [ :view | self update: view ].!

attachView: aView
  self validate: aView as: #view.
  self views add: aView.!

detachView: aView
  self views remove: aView.! !

!SubjectView methodsFor: 'supporting views' !

update: aView
  self draw: aView.!

draw: aView
  aView showValue: self getValue.!

setSubject: aSubject
  self validate: aSubject as: #subject.
  self subject: aSubject.
  self views == self ifTrue: [ self views: Set new ].! !

"sample use of the framework"
SubjectView new
  setSubject: ValueHolder new;
  attachView: BarGraphView new;
  attachView: DialGaugeView new;
  attachView: PercentageView new;
  setValue: 75.!
```

Listing 4: ButtonGroup Contract

```
"The following example is derived from the refinement of the
SubjectView contract called ButtonGroup on page 173 of [HHG90]."

SubjectView subclass: #ButtonGroup
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''!

!ButtonGroup class methodsFor: 'validating roles'

view
  ^ #( value chosen: )! !

!ButtonGroup methodsFor: 'supporting buttons' !

select: aButton
  self setValue: aButton value.!

update: aButton
  self getValue = aButton value
        ifTrue: [ self choose: aButton ]
        ifFalse: [ self unChoose: aButton ].!

choose: aButton
  aButton chosen: true.!

unChoose: aButton
  aButton chosen: false.! !
```

Listing 5: SubPaneMediators

```
Framework subclass: #SubPaneMediator
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''!
```


fact, by making inheritance decisions too soon you can blind yourself to the opportunity to use inheritance in a much better way. This constraint suggests that you should make inheritance decisions only after the entire system is completed.

SOLUTION

If one of the objects has a superset of the other object's variables, make it the subclass. Otherwise, make a common superclass. Move all of the code and variables in common to the superclass and remove them from the subclasses.

EXAMPLE

It is difficult to come up with an example of inheritance that isn't totally obvious. The problem is that before you see it, you can't imagine it, and after you see it, you can't imagine it any other way. So, if this example seems contrived, don't worry, your own problems will be much harder.

Here is an example in VisualWorks I ran across a couple of months ago. I had Figure1, a subclass of VisualPart. It had to be dependent on a several other objects, and it had to delete those dependencies when it was released.

```
Class: Figure1
    Superclass: VisualPart
    Instance variables: dependees

Figure>>initialize
    dependees := OrderedCollection new
```

Rather than use the usual addDependent: way of setting up dependencies, I implemented a new message in Figure1 called dependOn:.

```
Figure1>>dependOn: anObject
    dependees add: anObject.
    anObject addDependent: self
```

When the figure goes away, it needs to detach itself from everyone it depends on.

```
Figure1>>breakDependents
    dependees do: [:each | each removeDependent: self].
    super breakDependents
```

Then I created a Figure2. To get it up and running quickly I just copied the three methods above to Figure2 and set about programming the rest of it.

It was when I went to create Figure3 that I decided to take a break and clean up. I created DependentFigure as a subclass of VisualPart, gave it the variable dependees and the three methods above, made Figure1 and Figure2 subclasses of it, deleted their implementations of initialize, dependOn: and breakDependents, and then implemented Figure3.

OTHER PATTERNS

While you are factoring the code is often a good time to apply Compose Methods so you can move more code into the superclass.

CONCLUSION

I have presented a pattern called Factor a Superclass as an alternative to Separate Abstract from Concrete for creating inheritance hierarchies. Using Factor a Superclass, you will end up with superclasses that have more state. I'm not sure if this is a good thing or not. On the plus side, you will probably be able to share more implementation. On the minus side, you may find yourself applying the pattern several times to get the final result. You might factor two classes to get a third, then notice that once you look at the world that way you can factor the superclass with a previously unrelated class to get a fourth, and so on.

Beware of juggling inheritance hierarchies too much. You can waste lots of time factoring code first one way, then another, and find that in the end you aren't that much better off than you were when you started. Objects can survive less-than-optimal inheritance much better than they can encapsulation violations or insufficient polymorphism. Most expert designers agree that great inheritance hierarchies are only revealed over time. Make the changes that you can see are obvious wins, but don't worry about getting it instantly, absolutely right. You are better off getting more objects into your system so you have more raw material from which to make decisions. ▣

Kent Beck has been discovering Smalltalk idioms for eight years at Tektronix, Apple Computer, and MasPar 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, 408.338.4649 (voice), 408.338.3666 (fax), or 70761,1216 on Compuserve.

THE BEST OF comp.lang.smalltalk

Alan Knight

Extending the environment (part 1)

The Smalltalk development environment is excellent in many ways, but stagnant. The basic tools haven't changed much from when I first used Apple Smalltalk-80 on a Lisa in 1986. At that time Smalltalk and LISP systems led the way in interactive development environments. Now these environments exist for many languages, some of them very competitive with Smalltalk.

To be fair, there have been great improvements in some areas, mostly in the area of add-on products. These include GUI builders, team programming tools, profilers, and database interfaces. The basic tools—the browsers, inspectors and the debugger—remain almost unchanged. This is not because they defy improvement.

Fortunately, one of Smalltalk's strengths is the ease with which it can be customized and extended. In this column, the first of two parts, I'll discuss some simple extensions to these tools. Part two will look at some of the packages available that make more substantial changes. The main focus will be on ideas or on code available over the net rather than commercial products which are better covered in a product review.

AREN'T IMPROVEMENTS THE VENDOR'S JOB?

Ideally, users shouldn't have to write or acquire extended tools. The development environment is a strong selling point for Smalltalk, and one might expect the vendors to put some effort into improving it. From the vendor's point of view, however, there are good reasons not to change the environment.

- *Backward compatibility.* Everybody gets annoyed when system code changes. If users don't think the changes are worth breaking their code for, they'll be upset.
- *Disagreement.* Any vendor-imposed changes to the environment will be unpopular with some users, and questionable changes run the risk of a backlash rivaling that was received by the New Coke.
- *Priorities.* Vendors have limited resources, and are kept very busy developing new products and fixing the major problems with existing ones. The base environment isn't bleeding too badly, so resources go elsewhere.
- *Lack of competition.* With the recent growth in Smalltalk's popularity, many users are new to the language and come from areas such as mainframe COBOL or 4GL development. They're still too dazzled by the very idea of an incremental development environment to complain about its deficiencies. Competition from other languages isn't strong enough yet to inspire changes. The most likely source of improvements may be new Smalltalk vendors who need to worry more about carving a niche than backward compatibility.
- *Extensibility.* There are relatively few complaints about the environment, because any user with sufficient time and skill can change it to suit themselves.

IT'S UP TO YOU

You can't count on the vendors for improvements, so it's up to you to take responsibility for your own development environment. You don't have to rewrite the debugger, but don't be afraid to make changes or to explore the changes others have made.

At this point, careful readers may recall my March/April 1993 column, where I urged great caution in making system changes. This appears to be a contradiction, but it's really just a trade-off. To be sure, there are risks in changing the system. New releases or add-on products will need to be checked more carefully for conflicts and small mistakes can destroy an image. Frequent back-ups are in order.

On the other hand, changing the browsers or inspectors is much less risky than changing deep system components such as the compiler or the process scheduling mechanisms. Even with the risks, the increased productivity can be well worth the trouble. As always, it's best to limit changes in system methods to small "hooks" that call your own code. This helps minimize the problems with new releases.

WHAT NEEDS CHANGING

Development environments are a religious issue, and everyone has a different opinion on the perfect environment. Nevertheless, here's a short wish list of ideas. Note: Not all these ideas have been implemented, and if they have, the author is not necessarily in a position to distribute the code. The best place to look for code is the Smalltalk ftp archives (st.cs.uiuc.edu or mushroom.cs.man.ac.uk), where the authors have gone to the trouble of cleaning things up and releasing them to the public. Code written for personal use often requires significant effort to adapt and separate from other extensions.

This column mentions extensions from three different peo-


```
!Framework methodsFor: 'defining roles' !

addRolesNamed: roleNames
  roleNames do: [ :roleName | self for: roleName use: nil ].!

for: roleName use: anObject
  (self binders includes: roleName) ifFalse: [
        ^parts at: roleName put: anObject ].
  self
        perform: (self binderFor: roleName)
        with: anObject.!

when: eventName do: aBlock
  self for: eventName use: aBlock.! !

!Framework methodsFor: 'binding components' !

resolveRoles
  | framework |
  parts associationsDo: [ :model |
        framework := model value.
        (framework isKindOf: Framework) ifTrue: [
                framework name: model key.
                framework resolveRolesFrom: parts ] ].
  self validateParts.!

resolveRolesFrom: partsCatalog
  | part |
  self unresolvedRoleNames do: [ :roleName |
        part := partsCatalog at: roleName ifAbsent: [ nil ].
        self for: roleName use: part ].
  self validateParts.!

unresolvedRoleNames
  ^parts keys select: [ :roleName |
        (self partNamed: roleName) isNil ]! !

!Framework methodsFor: 'triggering events' !

notify: partName that: eventName
"Answer the result of notifying the named part that eventName
occurred."
  ^(self partNamed: partName)
        notifyThat: eventName!

notify: partName that: eventName with: argument
"Answer the result of notifying the named part that eventName
occurred."
  ^(self partNamed: partName)
        notifyThat: eventName
        with: argument!

notify: partName that: eventName withAll: arguments
"Answer the result of notifying the named part that eventName
occurred."
  ^(self partNamed: partName)
        notifyThat: eventName
        withAll: arguments! !

!Framework methodsFor: 'translating events to messages' !

respondsTo: selector
  (super respondsTo: selector) ifTrue: [ ^true ].
  (parts respondsTo: selector) ifTrue: [ ^true ].
  ^false!

notifyThat: eventName
"Answer the result of performing eventName, or the receiver if
eventName has not been implemented."
  ^self ifUnderstoodPerform: eventName!

notifyThat: eventName with: argument
"Answer the result of performing eventName, or the receiver if
eventName has not been implemented."
  ^self ifUnderstoodPerform: eventName with: argument!

notifyThat: eventName withAll: arguments
"Answer the result of performing eventName, or the receiver if
eventName has not been implemented."
  ^self ifUnderstoodPerform: eventName
        withAll: arguments! !

!Framework methodsFor: 'validating role services' !

canUse: part as: roleName
  self class ifUnderstood: roleName do: [
        ^part respondsToAll:
                (self class perform: roleName) ].
  ^true!

validate: part as: roleName
  | services |
  (self canUse: part as: roleName) ifTrue: [ ^self ].
  services := self class perform: roleName.
  services := part servicesRejectedFrom: services.
  ^self error:
        'Supplied ', roleName storeString,
        ' cant respond to ', services first storeString!

validateParts
  parts associationsDo: [ :each |
        self validate: each value as: each key ].! !

!Framework methodsFor: 'binding components - private' !

binderFor: roleName
"Answer the selector that can be used to bind a component to
roleName."
  ^( roleName, ':' ) asSymbol!

binders
"Answer all the selectors that can be used to bind the components of a
framework subclass."
  | supers methodNames |
  methodNames := Set new.
  supers := self class allSuperclasses removeLast; yourself.
  supers size > 0 ifTrue: [ supers removeLast ].
```

```
notifyThat: eventName
"Do nothing, as nil has no dependents."!

notifyThat: eventName with: argument
"Do nothing, as nil has no dependents."!

notifyThat: eventName withAll: arguments
"Do nothing, as nil has no dependents."! !

IdentityDictionary subclass: #SmartDictionary
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: '' !

!SmartDictionary methods !

respondsTo: selector
"Answer whether the receiver can respond to the message selector."
  | colons |
  (super respondsTo: selector) ifTrue: [ ^true ].
  (self includesKey: selector) ifTrue: [ ^true ].
  colons := selector occurrencesOf: ($:).
    colons = 1

doesNotUnderstand: aMessage
"If the receiver can handle aMessage selector, do so. Otherwise, treat
aMessage like super would."
  | name |
  name := aMessage selector.
  (self respondsTo: name) ifFalse: [
        ^super doesNotUnderstand: aMessage. ].
  "handle getter."
  (self includesKey: name) ifTrue: [ ^self at: name ].
  "handle setter."
  name := name asString copyWithout: ($:).
  ^self at: name asSymbol
        put: aMessage arguments first.! !
```

Listing 2: Framework class.

```
Object subclass: #Framework
  instanceVariableNames: 'name parts '
  classVariableNames: ''
  poolDictionaries: '' !

!Framework class methodsFor: 'creating instances' !

assemble: frameworkName from: parts
  ^self new
        name: frameworkName;
        parts: parts;
        resolveRoles!

new
  ^super new initialize! !

!Framework methodsFor: 'initializing - releasing' !

initialize
  name := nil.
  parts := SmartDictionary new.!

release
  | objects |
  objects := self parts.
  parts := SmartDictionary new.
  objects do: [ :each | each release ].
  ^super release! !

!Framework methodsFor: 'accessing components' !

name
  ^name!

name: partName
  name := partName.!

partNamed: partName
  ^self partNamed: partName ifNone: [ nil ]!

partNamed: partName ifNone: aBlock
  | part |
  part := parts at: partName ifAbsent: [ ^aBlock value ].
  part isNil ifTrue: [ ^aBlock value ].
  ^part!

partNames
  ^parts keys!

parts
  ^parts values!

parts: partsCatalog
  parts := partsCatalog.! !

!Framework methodsFor: 'assembling frameworks' !

bestRoleNameFor: part
"Answer the roleName that best fits the part, or nil."
  | roleName roleSize roleServices |
  roleSize := 0.
  roleName := nil.
  self class roleNames do: [ :each |
      self class ifUnderstood: each do: [
          roleServices := self class perform: each.
          roleServices size > roleSize ifTrue: [
              (self canUse: part as: each) ifTrue: [
                  roleName := each.
                  roleSize := roleServices size ] ] ] ].
  ^roleName!

useBestRoleFor: part
  | roleName |
  roleName := self bestRoleNameFor: part.
  roleName isNil ifFalse: [
        self for: roleName use: part ].! !
```


ple on the net. **Deeptendu Majumder** (dips@cad.gatech.edu) has released his extensions up in a package called ISYSE, available from the archives.

Bruce Samuelson (bruce@ling.uta.edu) may get around to cleaning up and releasing his code, but is not in a position to do so at this time. **Gene Golovchinsky** (golovch@ie.toronto.edu) hasn't packaged his extensions, but is willing to be pestered about them.

Automatically writing access methods

One of the most common system extensions is a mechanism to generate access methods for instance variables. These methods aren't difficult to write by hand, but they occur so frequently that a tool can be very convenient.

It's important that the tool be selective. Not all variables should have access methods (or some of them should be clearly marked private, depending on your philosophy) so the user must be able to select which methods to generate. The tool should also provide documentation in the method. The user should be able to (if not forced to) provide information on the type of the variable and its purpose. This information should already be in the class comment, but it doesn't hurt to duplicate it. A really sophisticated tool would check the class comment for the information and update it if necessary.

Find class

I use the "Find class" feature very frequently, especially in Digitalk dialects. Unfortunately, the basic Digitalk implementation is brain-dead, and the ParcPlace one, while better, still doesn't do what I want.

- Ignore case. This is much faster and more convenient. (Is it Filename or FileName?)
- If the name matches a class (e.g., set), go directly to it without presenting a useless list of one class to choose from. In general, I prefer tools that can skip over lists with only one item.
- If the name doesn't match a class, append a wildcard and present a list of those it matches (e.g., sett gives me a list of #(Settee Setter Settlement).
- If I explicitly type a wildcard, always give me the list (e.g., set* gives #(Settee Setter Settlement)).

Smalltalk/V's debugger

If you've used both Smalltalk-80 and Smalltalk/V, one of the most frustrating things about V is its debugger. To the untrained eye, both debuggers are very similar, and in fact V offers the nice additional feature of breakpoints. The problem is that when evaluating an expression inside the debugger, V evaluates it as a method in **self** (the receiver of the current message), *not* the context of the current method. In the Smalltalk-80 debugger you can highlight any text in the current method and evaluate it. In the Smalltalk/V debugger this only works if the text doesn't reference method arguments or locals.

The most irritating thing about this problem is that I don't know how to fix it. Digitalk hides the source to their compiler, and although I've come up with a few bizarre ideas that might work, I've never had time to really work on it. If anybody has a fix for this, please let me know.

Browsing inherited methods

I don't know how many requests I've seen for a for a browser that shows all methods in a class, including inherited methods. The basic functionality is very simple, and the real problem is providing a good user interface. ParcPlace does provide this capability with the FullBrowser, but it's a poor implementation and only available in the APOK add-on package. It's a good example of why we might not want the vendors deciding for themselves how to improve the environment. Most of the extended environments described in part 2 provide this capability in some form.

Resizing panes

Bruce Samuelson describes a useful feature to augment the browser with:

> **...buttons for resizing browser windows horizontally and vertically, and reproportioning the line separating the upper panes from the method**

This is an increasingly common feature in user interfaces, and one that can be very useful. Smalltalk/V Mac has a convenient "zoom" feature that makes the text editing area fill the entire window, but this would be more flexible.

Gene Golovchinsky writes:

> **I would like to see more buttons on the screen for common commands rather than entries in pop-up menus. I invariably pick the wrong one, or keep moving between copy, paste, and accept. Then I accidentally pick cancel, and have to repeat the whole process again!**

I'm not sure we want to add too many buttons, but a few in the right place would be nice. Certainly, it's much nicer having buttons in the debugger for single stepping than having to use a pop-up menu. For operations like cut and paste I prefer to have keyboard short-cuts.

Renaming classes in Smalltalk/V

Smalltalk/V still doesn't support renaming classes or changing the definition of classes with instances. It shouldn't be that hard to implement, and I believe the capabilities are available as part of their Team/V package. Why is such a basic capability bundled into a team programming tool and not in the base image? Only Digitalk can tell.

COGNITIVE OVERLOAD

While all of the above are useful, they are only minor improvements. There are more general issues that need to be addressed. **Deeptendu Majumder** raises the issue of cognitive overload in the Smalltalk environment:

One thing that irritates me more and more these days is how my screen gets out of control with a multitude of windows . . . . I sometimes wonder if there is some kind of study . . . about determining the most suitable ST programming environment . . . . I sometimes *very strongly* feel the environment can be "smarter" about . . . reducing the cognitive overload *and* maintaining easily identifiable cues about what info is available only for a mouse click.

Controlling windows

The largest single factor in cognitive overload must the number of windows Smalltalk produces. I usually have 10 to 20 windows open simultaneously and I'm sure I get as high as 50 now and then. With this many windows, it's vital to have mechanisms to control the complexity.

Craig Latta (latta@xcf.berkeley.edu) writes:

I find that simply having a good window manager goes a long way toward reducing the cognitive load. The main problem I would have otherwise is with hordes of windows crowding the screen, and subsequently losing track of particular windows. Things like icon managers (as in 'twm' on X platforms) reduce this problem significantly.

A good window manager and a large screen are vital elements for Smalltalk work. One technique I use is to make use of window and icon positions. Certain windows (e.g., the system transcript, a workspace with useful expressions, my list of things to do) are always open, and I make a point of always keeping them in the same place. I also try to keep their icons in standard places, but not all window managers maintain the position of icons (MS-Windows doesn't).

> "Writing Smalltalk code is akin to authoring hypertext"

Another technique is to put more information into window titles. By hooking into the browser selection mechanism, the window title can be made to indicate the current class and method. This makes navigating among icons easier, and can also be used with window managers that allow you to find windows by title. With a bit more effort, it should be possible to change the window icon to convey more information.

If your window manager doesn't manage windows and icons well, it's possible to make up some of the difference in Smalltalk. Gene Golovchinsky writes:

I added an entry to the Launcher menu that displays a list of all current Smalltalk windows, and indicates the minimized ones. If I pick from this menu, it raises that window. Just today I saw that something similar is available in the archives!

Reducing the number of windows

Managing windows is all very well and good, but do we really need all those windows in the first place? Jaap Vermeulen (jaap@sequent.com) doesn't think so. He writes:

With new tools to replace the browsers that allow better indexing, searching, shortcuts, and backtracking, you might need fewer windows. Finally, if the inspectors and debugger would become a little smarter and not throw up windows all over the place, we really would start talking.

Inspectors are one of the worst culprits in creating excess windows. A tool that allowed graphical inspecting of many objects at once, following links between them, could reduce this considerably. There is a simple tool of this type included with the HotDraw application framework. I believe First Class Software (408.338.4649 (voice), 408.338.3666 (fax), or 70761.1216 on CompuServe) has a graphical inspecting tool for Smalltalk/V.

Too many browser operations spawn a new window in which to present their results. The only concept of backtracking is to go back to the window you started the operation from. For operations like **senders**, this is simple to change and makes the function easier to use. Gene Golovchinsky writes:

I've augmented the MethodListBrowser to add the ability to add a specific method to the list. It works like the Messages menu item, but instead of spawning a new window, it adds the entry to the list. If there is more than one item, it prompts for the one to add. I find this tool handy for traversing long chains of message sends and keeping them all in one place.

Unfortunately, it's not so easy to reduce the number of windows generated by some of the other operations.

HYPERTEXT MECHANISMS

Gene Golovchinsky writes:

Writing Smalltalk code is akin to authoring hypertext; perhaps some insight can be gained from perusing that literature. Along those lines, this environment seems like an ideal vehicle for implementing all sorts of hypertext behavior. In fact, the existing browsers have many of these features already.

Indeed, Smalltalk browsing shares many characteristics with hypertext browsing and suffers many of the same problems. There's an enormous amount of information, only a small part of which is relevant at any given time, and it's easy to become lost in the irrelevant.

Messages

Many browser improvements are intended to quickly find relevant information while avoiding that which is not relevant. If you can follow a link directly to what's important, you don't need as many windows open looking for it.

One such feature is the messages menu item mentioned above. This allows you, when browsing a method, to find im-

Listing 1. Framework Support.

```
"The following code extends the baseline Smalltalk classes to support
certain aspects of framework assembly, event handling, and role
validation."

!Collection methods !

assembleAs: frameworkClass
"Answer a new framework assembled from the receiver."
  | framework |
  framework := frameworkClass new.
  self do: [ :each | view useBestRoleFor: each ].
  ^framework resolveRoles! !

!Object methodsFor: 'accessing named dependents' !

dependentNamed: name
"Answer the named dependent, or nil."
  ^self dependentNamed: name ifNone: [ nil ]!

dependentNamed: name ifNone: aBlock
"Answer the named dependent, or evaluate aBlock."
  ^self namedDependents
      detect: [ :d | d name = name ] ifAbsent: aBlock!

namedDependents
"Answer any named dependents attached to the receiver."
  ^self dependents
      select: [ :each | each respondsTo: #name ]! !

!Object methodsFor: 'performing optional behaviors' !

ifUnderstood: selector do: aBlock
"Evaluate aBlock if the receiver understands selector."
  ^(self respondsTo: selector)
      ifTrue: aBlock
      ifFalse: [ self ]!

ifUnderstoodPerform: selector
"Answer the result of the selected method, or the receiver."
  (self respondsTo: selector) ifFalse: [ ^self ].
  ^self perform: selector!

ifUnderstoodPerform: selector with: argument
"Answer the result of the selected method, or the receiver."
  (self respondsTo: selector) ifFalse: [ ^self ].
  ^self perform: selector with: argument!

ifUnderstoodPerform: selector withAll: arguments
"Answer the result of the selected method, or the receiver."
  (self respondsTo: selector) ifFalse: [ ^self ].
  ^self perform: selector withArguments: arguments! !

!Object methodsFor: 'notifying dependents of events' !

notify: name that: eventName
  ^(self dependentNamed: name)
      notifyThat: eventName!

notify: name that: eventName with: argument
  ^(self dependentNamed: name)
      notifyThat: eventName
      with: argument!

notify: name that: eventName withAll: arguments
  ^(self dependentNamed: name)
      notifyThat: eventName
      withAll: arguments!

notifyThat: eventName
"Answer the final result of notifying all the dependents that eventName
occurred."
  | result |
  self dependents do: [ :d |
      result := d notifyThat: eventName ].
  ^result!

notifyThat: eventName with: argument
"Answer the final result of notifying all the dependents that eventName
occurred."
  | result |
  self dependents do: [ :d |
      result := d notifyThat: eventName
          with:argument ].
  ^result!

notifyThat: eventName withAll: arguments
"Answer the final result of notifying all the dependents that eventName
occurred."
  | result |
  self dependents do: [ :d |
      result := d notifyThat: eventName
          withAll: arguments ].
  ^result! !

!Object methodsFor: 'responding to requests' !

respondsToAll: symbolSet
"Answer whether the receiver responds to all of the messages in
symbolSet."
  symbolSet do: [ :each |
      (self respondsTo: each) ifFalse: [ ^false ] ].
  ^true!

servicesRejectedFrom: symbolSet
"Answer those service requests from symbolSet to which the receiver
does not respond."
  ^symbolSet reject: [ :each | self respondsTo: each ]!

value
"Answer the receiver."
  ^self! !

!UndefinedObject methodsFor: 'catching dependents access' !

namedDependents
"nil has no dependents."
  ^Array new!
```

through such an implicit "second-class" framework can be difficult. However, these patterns of interaction can be captured and reused explicitly by framework classes. Because the message flow is more explicit in framework classes, they are much easier to understand.

As noted perviously, good class hierarchies tend to be deep and narrow. The hierarchies created by framework classes tend to be deep, narrow, and *thin*. The methods themselves tend to be small (thin), because they coordinate only the interactions between the objects that participate in the framework.

Many object designers have claimed that frameworks are difficult to find. Actually, frameworks are not hard to find at all! They simply have not been noticed much. They tend to be like thin oils that lubricate the meshings of larger objects. Any pattern of interactions between objects may be captured as a framework. However, the resulting framework may be so specialized that it is better to leave the interactions built into the collaborating classes. Frameworks serve best when they capture and factor out the semantics of event-driven interactive systems.

Sometimes it is expedient during prototyping to develop a system that is closely coupled. After completing the prototype, some parts of the design can be revisited and the coupling loosened for better reusability. Loosely coupled objects tend to be more reusable and more resilient to design and system evolution. Framework classes provide a new option for refactoring through decoupling.

FUTURE WORK

The current implementation of the Framework superclass uses a simple collection of method names for role validation. It would be better if each role were defined using a specification object, in particular an object type. Object types use method signatures to specify the types of each argument and the method result. When these specification objects become available, framework role validation can evolve to use them. Object types will provide better constraints to qualify components for roles.

CONCLUSION

This article has presented a new view of object frameworks: How framework classes can simplify the design of component classes by factoring out the behavior found in interactive systems. Component objects become simply clients and/or service providers, reducing or eliminating the additional responsibilities of complex coordination between objects. In addition to simplifying existing components, refactoring may create new components. Such refactoring improves the reusability of all the components that form a system and creates reusable framework objects. ▣

Acknowledgments

Several individuals inspired me with their interest and thoughtful critiques during the evolution of these ideas. Special thanks to Jean-Francois Cloutier, Tracy Tondro, Oleg Arsky, and Jim Carlstedt.

References

1. Krasner, G.E., and S.T. Pope. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80, JOURNAL OF OBJECT-ORIENTED PROGRAMMING 1(3):26–49, 1988.
2. Shan, Y-P. An event-driven model-view-controller framework for Smalltalk, OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES, AND APPLICATIONS CONFERENCE, ACM, New Orleans, LA, 1989.
3. Shan, Y-P. MoDE: A UIMS for Smalltalk,. OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES, AND APPLICATIONS CONFERENCE, ACM, Ottawa, ONT, 1990.
4. Sullivan, K.J., and D. Notkin. Reconciling environment integration and component independence, TRANSACTIONS ON SOFTWARE ENGINEERING, ACM, Ottawa, ONT, 1990.
5. Helm, R., I.M. Holland, and D. Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems, OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES, AND APPLICATIONS CONFERENCE, ACM, Ottawa, ONT, 1990.
6. Wilkerson, B. How to design an object-based application, DEVELOP, Apple Computer, Cupertino, CA, April, 1990.
7. Wirfs-Brock, R., and R.E. Johnson. A survey of current research in object-oriented design. COMMUNICATIONS OF THE ACM 33(9):104–124, 1990.
8. Wirfs-Brock, R., and B. Wilkerson. Object-oriented design: A responsibility-based approach, OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES, AND APPLICATIONS CONFERENCE, ACM, New Orleans, LA, 1989.
9. Wirfs-Brock, R., B. Wilkerson, L. Wiener. DESIGNING OBJECT-ORIENTED SOFTWARE, Prentice Hall, Englewood Cliffs, NJ, 1990.
10. Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy, W. Lorensen. OBJECT-ORIENTED MODELING AND DESIGN, Prentice Hall, Englewood Cliffs, NJ, 1991.
11. Opdyke, W.F. Refactoring object-oriented frameworks, PhD thesis, University of Illinois at Urbana-Champaign, 1992.
12. Cook, W.R. interfaces and specifications for the Smalltalk-80 collection classes, OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES, AND APPLICATIONS CONFERENCE, ACM, Vancouver, BC, 1992.
13. Boyd, N. Modules: Encapsulating behavior in Smalltalk, THE SMALLTALK REPORT 2(5), 1993.

Nik Boyd is a Principal Member of the Technical Staff at Citicorp Transaction Technology in Santa Monica, CA. His research interests include instance-centered and class-centered object systems, as well as tools and techniques that support object-oriented software engineering. Nik may be contacted via email at 74170.2171@compuserve.com or through the American Information Exchange (AMIX).

plementors or senders of any of the messages sent by that method. The messages sent become hypertext links.

One problem is that the number of methods found can be too large to work with. Thus, it's useful to restrict the methods considered. One way is to allow "local" senders/implementors, selecting only methods within the current class or perhaps within its sub/superclasses.

Bruce Samuelson has another mechanism:

> **…'my senders', 'my implementors' which only look at the changes file…**

Also, we may want to browse a method that isn't sent from the current message, or we may be in a text editor instead of a browser. **Gene Golovchinksy** describes a menu item that opens a browser on the class or method named by the currently selected text. I have a similar extension, but I separate the browse/senders/implementors/class references behavior and use keyboard shortcuts to invoke them. Keyboard shortcuts are a little faster, and work in workspaces as well as browsers, but are less mnemonic and not as flexible.

Operating on text is a nice feature, but one that works best for zero- or one-argument messages. Multi-keyword messages don't usually occur in text in the right form. It should be possible to use the Smalltalk parser to extract possible message names, but I haven't tried this.

Deeptendu Majumder added a feature for finding implementors of a method whose name is not known. The base image allows wildcard searches on method names, but force a choice from a menu of possible names without seeing implementations.

> **…all I did was add an extra list to the browser that grabs all those things that otherwise show up in the menu. When I am not sure exactly which method I am looking for, I can select entries from this list one after another and browse their various implementations. I can then change the selection template from within the list and grab a whole new set of message names.**

Searching for strings

The link you need may not be the name of the method or a message that it sends. Just today I wanted to search for a method that didn't send a particular message, but contained the name of that message in a comment. I had previously commented out that message send, closed the window, and forgotten the method name. **Bruce Samuelson** writes of a feature he implemented:

> **…search for a string (e.g., open:) in methods and class comments. This can operate on…categories, classes, or protocols. This is useful for maintaining comments and for finding code for which standard searches break down.**

Lost in hypertext tools

All the mechanisms listed above are valuable tools for searching. Unfortunately, if we implemented them all in a single image I suspect users would merely find themselves lost in hypertext mechanisms instead of (or as well as) lost in the code. As **Deeptendu Majumder** writes:

> **There are so many small enhancements that can be done that I found it is not very productive to undertake the effort without a serious study of overall needs rather than trying to attack small segments of the problem.**

Next month, we'll examine some more radical extensions that replace the basic tools instead of patching or adding a few features.

ERRATA

In the June 1993 column I published code for testing dictionary performance under ObjectWorks\Smalltalk release 4.0. Unfortunately, I didn't test this code adequately, and Bruce Samuelson, the author, has pointed out that, due to changes, this code does not work with release 4.1 or VisualWorks. There are two problems. First, the way hashing is done has changed, so the results will be in error. Second, the method sortedElements has been removed, so the method will produce a walkback. A new version, which will also work with other hash table classes, is available from the Smalltalk archives at st.cs.uiuc.edu. ▣

Alan Knight works for The Object People. He can be reached at 613.225.8812 or by email as knight@mrco.carleton.ca.

continued from page 6

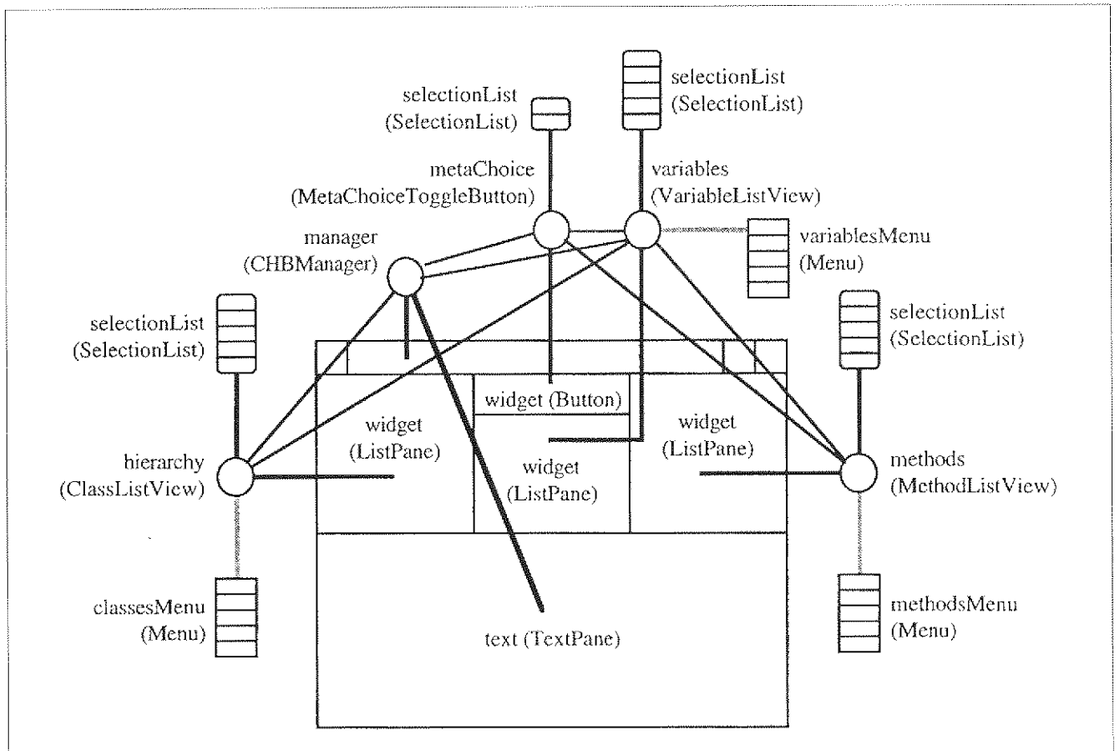


Figure 2. CHB frameworks.

dered dictionary key. SelectionLists also notify their dependent mediators when their list or selection changes:

```
"from within #list:"
self notifyThat: #listChanged.

"from within #select:"
self notifyThat: #selectionChanged.
```

Listing 5 shows the code for the SubPaneMediator classes. The kinds of SubPaneMediators that use SelectionLists include those depicted in the following hierarchy:

```
Object
  Framework
    SubPaneMediator
      ListItemChooser
        ListViewer
        ListButton
          MenuButton
          ToggleButton
```

The ListItemChooser class manages the interactions between a SelectionList and a SubPane (GUI widget). The ListViewer class manages the interactions between a SelectionList and a ListPane. The ListButton classes manage the interactions between a SelectionList and a Button in two varieties. The MenuButton class pops up a menu of the list items when clicked, allowing one of the items to be selected. The ToggleButton cycles through the list of items, showing the next item description on the button face.

Now, consider how these small framework classes might be used to refactor a browser such as the Smalltalk/V ClassHierarchyBrowser (CHB). The CHB has five subpanes: a class hierarchy ListPane, a variables ListPane, a methods ListPane, a RadioButton group, and a TextPane.

For this discussion, we will replace the RadioButton group with a specialization of the ToggleButton. This MetaChoiceToggleButton framework will use a two item list: #(class instance) for selecting either class methods or instance methods.

For each of the ListPanes, we specialize the ListViewer framework with ClassListViewer, VariableListViewer, and MethodListViewer frameworks. Each of these small frameworks serves as the owner for their respective subpanes. As such, they accrete the behavior from the CHB related to those panes, including menus, list maintenance, item selection, and propagation of notifications and changes throughout the overall framework network (see Figures 1 and 2).

This brief outline indicates how such refactoring can proceed. However, note that further evolution and improvements can be made through additional refactoring and framework creation. In the end, the responsibility of the browser class can be reduced to assembling a network of objects that together produce the overall browser behavior.

TUNING COMPONENT COUPLING

The Framework superclass uses loose coupling as a technique for achieving component integration and coordination. The implementation suggested in this article makes use of a kind of Dictionary to bind framework participants into their roles. This technique of loose binding allows frameworks to be evolved and extended quickly through several iterations.

Although this technique requires little in the way of overhead, a small amount of performance can be lost when the role participants are resolved dynamically. A number of options exist for tuning the performance of frameworks built using these techniques.

The Framework class uses a class named SmartDictionary (see Listing 1). In addition to the messages understood by IdentityDictionary, SmartDictionary responds to the typical accessor idioms:

```
componentName "getter"

componentName: anObject "setter"
```

These protocols are supported by overriding the #respondsTo: and #doesNotUnderstand: methods. These protocols are also supported by the Framework class. In addition to this implicit form of component access, the Framework class supports the following form of indirect access:

```
componentName "indirect getter"
  ^self partnerNamed: #componentName!

componentName: anObject "indirect setter"
  self
      for: #componentName
      use: anObject.!
```

This support for the dynamic binding of roles can be replaced by ordinary instance variables and their accessors. However, in order to gain the benefits of rapid design evolution, this should be done (if done at all) only after the design of the framework class has stabilized.

```
componentName "direct getter"
  ^componentName!

componentName: anObject "direct setter"
  componentName := anObject.!
```

PARTICIPANT INTERACTION STYLES

One of the principal uses of any framework class is to mediate the interactions of its participants. Because participants are loosely coupled, the methods of a framework class have this peculiar aspect: Participants are *always* accessed through requests to self. So, some of the framework methods provide access to components or their state(s), while others translate events into actions.

The event handling methods of a framework class serve as templates that guide the exchange of information between the framework participants. The expressions used by these event handling methods generally fall into one of the following basic patterns:

```
eventName
"Request information or a change of state."
  ^self someComponent request

eventName
"Exchange information between components."
  self someComponent binaryKeyword:
      self anotherComponent request.

eventName
"Notify another participant (framework) that something happened
(translating the event name)."
  ^self
      notify: #frameworkX
      that: #somethingHappened

eventName
"Forward this event to another participant (framework)."
  ^self
      notify: #frameworkX
      that: #eventName
```

SPECIALIZING FRAMEWORKS

New frameworks can often be discovered when reusing existing ones. Sometimes it is more convenient to attach custom behavior to an existing framework rather than create a new framework subclass.

The Framework superclass supports the prototyping of new behavior by allowing the usage of blocks as components. When a message is redirected through #doesNotUnderstand:, the Framework superclass checks to see if a block has been defined to handle the message selector. If the framework can handle the message with a block, the block is evaluated with the message receiver and its arguments (if any).

After a new framework has stabilized, the developer may decide to create a new framework subclass, moving its specialized behavior from blocks into methods. When this occurs, the developer is faced with a decision: What should the scope of visibility for the new class be? Very general frameworks should probably be visible to the whole Smalltalk system. However, some frameworks should only be visible to the class(es) that need them. Module classes[13] can be used to hide specialized framework subclasses.

For example, in our consideration regarding browsers, we found that they will often need specialized frameworks for managing the interactions between the subpanes from which they are composed. Each of the ListItemChooser subclasses can be further specialized to create customized mediators that manage the overall interactions between the various subpanes that make up a browser. Rather than expose these specialized frameworks to the whole of Smalltalk, they can be hidden within the browser class if it is implemented as a module.

GENERAL OBSERVATIONS

Many patterns of interaction between objects in a system appear over and over again in other systems. Sometimes these patterns are formed into a loose composition of abstract classes like the MVC framework.[1] Following the flow of messages

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS MAIL PERMIT NO. 279 DENVILLE NJ

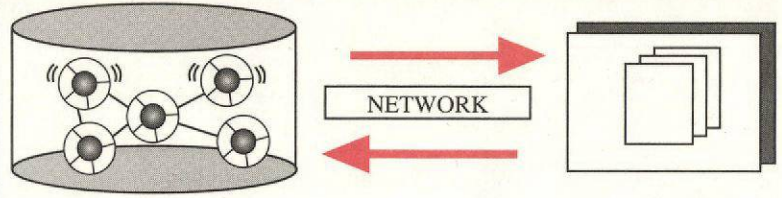
POSTAGE WILL BE PAID BY ADDRESSEE

The Smalltalk Report
Subscriber Services Dept SML
PO Box 3000
Denville NJ 07834-9821

If You Use Smalltalk, You Need GemStone.

GemStone is the ideal database environment for supporting Smalltalk applications. It is the only high-performance, production-ready ODBMS with a transparent Smalltalk interface.

- Maintain class hierarchies and execute Smalltalk methods directly in the server.
- Automatic, transparent translation of Smalltalk objects into GemStone.
- Cooperative client-server support.
- Smalltalk-based DDL/DML.
- High-performance, scalable, production-ready ODBMS.
- Integrated garbage collection of persistent Smalltalk objects.



❑ **YES! Send Me Complete Details On GemStone**

Name: _____ **Title:** _____

Company: _____

Address: _____

City: _____ **State:** _____ **Zip:** _____

Phone: _____ **Fax:** _____

1-800-243-9369

SERVIO

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 4362 SAN JOSE, CA

POSTAGE WILL BE PAIDBY THE ADDRESSE

SERVIO CORPORATION
2085 HAMILTON AVENUE
SUITE 200
SAN JOSE, CA 95125-9985

The Smalltalk Report

☞ **Provides objective & authoritative coverage on language advances, usage tips, project management advice, A&D techniques, and insightful applications.**

"If you're programming in Smalltalk, you should be reading **The Smalltalk Report**"

❑ **Yes, I would like to subscribe to The Smalltalk Report** Date_____

❑ 1 year (9 issues)
- ❑ Domestic $69.00
- ❑ Foreign $94.00

❑ 2 year (18 issues)
- ❑ Domestic $128.00
- ❑ Foreign $178.00

Method of Payment

❑ Check enclosed (payable to **The Smalltalk Report**)
❑ Bill me
❑ Charge my: ❑ Visa ❑ Mastercard ❑ Amex
Card No. _____
Exp. Date _____
Signature _____

Name _____
Title _____
Company _____
Address _____
City _____
State _____
Zip _____
Country _____
Phone _____

1. Which dialect of Smalltalk do you use:
- ❑ Smalltalk V
- ❑ Smalltalk-80
- ❑ Other _____

2. What is your involvement in software purchases for your department/firm:
- ❑ Recommend Need
- ❑ Specify Product
- ❑ Make Purchase
- ❑ None

3. Which operating system supports your software:
- ❑ UNIX
- ❑ DOS
- ❑ OS/2
- ❑ Windows
- ❑ Other _____

4. What is your company's primary business activity:
- ❑ Computer/Software Development.
- ❑ Manufacturing
- ❑ Financial Services
- ❑ Government/Military/Utility
- ❑ Educational/Consulting
- ❑ Other _____

5. For how long have you been using Smalltalk:
- ❑ Less than one year
- ❑ 1–3 years
- ❑ 3+ years

E3JG

A member of the Object Marketing Network

fax to 212/274-0646

SIGS PUBLICATIONS